**Tese apresentada à Divisão de Pós-Graduação do Instituto Tecnológico de Aeronáutica como parte dos requisitos para obtenção do título de Doutor em Ciência no Curso de Engenharia Eletrônica e Computação, Área de Informática.**

**Carlos Alberto Alonso Sanches**

# Algoritmos Paralelos para o Problema da Mochila

Prof. Dr. Nei Yoshihiro Soma
Orientador

Prof. Dr. Horácio Hideki Yanasse
Co-orientador

Prof. Dr. Homero Santiago Maciel
Chefe da Divisão de Pós-Graduação

Campo Montenegro
São José dos Campos – SP – Brasil
2003

# Algoritmos Paralelos para o Problema da Mochila

**Carlos Alberto Alonso Sanches**

Composição da Banca Examinadora:

| | |
|---|---|
| Prof. Dr. Celso de Renna e Souza | Presidente - IEC/ITA |
| Prof. Dr. Nei Yoshihiro Soma | Orientador - IEC/ITA |
| Prof. Dr. Horácio Hideki Yanasse | Co-orientador - LAC/INPE |
| Prof. Dr. Vakulathil Abdurahiman | IEC/ITA |
| Prof. Dr. Siang Wun Song | IME/USP |
| Prof. Dr. Celso Carneiro Ribeiro | DCC/UFF |

*Durante a elaboração deste trabalho, o autor recebeu apoio financeiro da FAPESP (processo 99/09483-5).*

**ITA**

*aos*

*meus pais*

*"... as almas têm um paralelismo tremendo, ainda que sejam diferentes."*

S. Josemaría Escrivá, 19-III-1975

## Agradecimentos

Como é difícil escolher palavras adequadas para esta página... Afinal, são muitas as pessoas que me ajudaram, não apenas na confecção deste trabalho, mas contribuindo em tantos aspectos imprescindíveis da minha formação.

Na absoluta impossibilidade de citá-las nominalmente (pelo simples fato de que são muitas), o mínimo que posso fazer em retribuição é expressar-lhes aqui a minha sincera e profunda gratidão.

No entanto, seria injusto se não destacasse meus pais, pois — aproveitando palavras que não são minhas — devo-lhes pelo menos 90% do que sou. A eles, que sempre me deram um grande exemplo de carinho e de dedicação, ofereço não somente esta tese, mas tudo o que ainda venha a fazer.

# Resumo

Esta tese melhora o *upper bound* de tempo e de espaço da resolução paralela do *Subset-Sum Problem* (SSP) — que é uma variante do Problema da Mochila — numa máquina PRAM SIMD CREW *(Parallel Random Access Machine; Single Instruction/Multiple Data; Concurrent Read/Exclusive Write)* nos dois paradigmas mais consagrados na literatura científica, isto é, tanto na abordagem através das **listas** como por **programação dinâmica**.

Com relação ao primeiro paradigma, é apresentada uma paralelização ótima e adaptativa do conhecido *algoritmo das duas listas* de Horowitz e Sahni [*JACM, 1974*] numa PRAM SIMD CREW de $p$ processadores: ela resolve o SSP de $n$ objetos em tempo $\mathcal{O}(2^{n/2}/p)$ e espaço $\mathcal{O}(2^{n/2})$, onde $1 \leq p < 2^{n/2}/n^2$. Como esse algoritmo seqüencial tem até hoje a melhor complexidade de tempo para a resolução do Problema da Mochila, então nosso algoritmo paralelo pode ser considerado, a partir de agora, como o melhor resultado teórico de toda a literatura.

Além disso, são apresentados três algoritmos paralelos adaptativos baseados no paradigma da programação dinâmica, que são os primeiros a resolverem o SSP de $n$ objetos e capacidade $c$ em tempo $o(nc/p)$ e espaço $\mathcal{O}(n + c)$ numa PRAM SIMD CREW de $p$ processadores. Eles melhoram as complexidades de tempo e de espaço do algoritmo de Lin e Storer [*JPDC, 1991*], que vinha sendo o mais eficiente até o momento.

## Abstract

The parallel solution for the *Subset-Sum Problem* (SSP) — which is a variation of the Knapsack Problem — is examined in this thesis to a PRAM SIMD CREW *(Single Instruction/Multiple Data; Concurrent Read/Exclusive Write)* for the two usual paradigms to the problem, *i.e.*, the **two-lists** enumeration scheme and **dynamic programming**.

For the *two-lists* enumeration scheme, it is suggested an optimal parallel adaptive algorithm of the well known Horowitz and Sahni algorithm [*JACM, 1974*] to a PRAM SIMD CREW with $p$ processors. The resulting parallel scheme solves the SSP with $n$ objects in a time bounded by $\mathcal{O}(2^{n/2}/p)$ and space $\mathcal{O}(2^{n/2})$, where $1 \leq p < 2^{n/2}/n^2$. Since the *two-lists* approach has the best known time complexity for the serial case to the present, the parallel version presented here might be the considered also the best bound for parallel environment counterpart.

Additionally, three parallel adaptive algorithms based on *dynamic programming* are also proposed and they are, to the best of the author's knowledge, the first ones to solve the SSP with $n$ objects and capacity $c$ in a time $o(nc/p)$ and space $\mathcal{O}(n + c)$. They were designed to a PRAM SIMD CREW with $p$ processors and the results configure an improvement in time and space in comparison with the best known parallel algorithm, the Lin and Storer [*JPDC, 1991*].

# Sumário

# Lista de Figuras

# Lista de Símbolos e Siglas

| SÍMBOLO OU SIGLA | SIGNIFICADO | PRIMEIRA OCORRÊNCIA |
|---|---|---|
| EREW | *Exclusive Read, Exclusive Write* | 19 |
| CREW | *Concurrent Read, Exclusive Write* | 19 |
| CRCW | *Concurrent Read, Concurrent Write* | 19 |
| ERCW | *Exclusive Read, Concurrent Write* | 20 |
| SISD | *Single Instruction stream / Single Data stream* | 21 |
| MISD | *Multiple Instruction stream / Single Data stream* | 21 |
| SIMD | *Single Instruction stream / Multiple Data stream* | 21 |
| MIMD | *Multiple Instruction stream / Multiple Data stream* | 21 |
| PRAM | *Parallel Random Access Machine* | 26 |
| KP | *Unbounded Knapsack Problem* | 31 |
| KP01 | *0-1 Knapsack Problem* | 30 |
| SSP | *Subset-Sum Problem* | 29 |
| SSPd | *Subset-Sum Decision Problem* | 32 |
| $T_s$ | Tempo gasto por algoritmo seqüencial | 25 |
| $T_p$ | Tempo gasto por algoritmo paralelo | 25 |
| $p$ | Número de processadores | 26 |
| $P_i$ | Processador de índice $i$ | 26 |
| $n$ | Número de objetos a serem acondicionados na mochila | 29 |
| $c$ | Capacidade da mochila | 30 |
| $v_i$ | $i$-ésimo objeto que pode ser acondicionado na mochila | 30 |
| $V$ | Conjunto dos $n$ objetos | 30 |
| $l_i$ | Lucro associado ao objeto $v_i$ | 30 |
| $w_i$ | Peso associado ao objeto $v_i$ | 30 |
| $w_{min}$ | Menor peso dentre os $n$ objetos | 30 |
| $w_{max}$ | Maior peso dentre os $n$ objetos | 30 |
| $W$ | Coleção dos pesos associados aos $n$ objetos | 30 |
| $X$ | Vetor binário com solução do KP01, SSP ou SSPd | 30 |
| $A, B$ | Listas de sub-soluções do KP01, SSP ou SSPd | 34 |
| $A_i, B_i$ | Blocos das respectivas listas de sub-soluções | 61 |
| $f_i(j)$ | Função de otimalidade para a resolução do KP01 | 38 |
| $F_i(c)$ | Vetor de lucro ótimo para a resolução do KP01 | 38 |

# Capítulo I

# Uma rápida introdução ao Paralelismo

Para descrevermos o âmbito do trabalho desta tese, que trata da elaboração e análise de algoritmos em um determinado modelo de máquina paralela, será preciso apresentar inicialmente uma série de conceitos básicos.

Apoiando-nos em alguns livros clássicos sobre Paralelismo [Akl89, JaJ92, Qui94, KGG94, Akl97, Leo01], trataremos rapidamente dos seguintes temas:

a) motivações para se pesquisar nessa área;

b) principais modelos de máquinas paralelas;

c) critérios de análise de algoritmos paralelos.

Além disso, apenas a título de esclarecimento, incluímos nesta introdução uma breve diferenciação entre Paralelismo e Sistemas Distribuídos, com a finalidade de caracterizar bem a área em que estaremos trabalhando.

## I.1 Motivações

Sabe-se que o desempenho dos computadores seqüenciais, isto é, das máquinas que executam uma única operação por vez, está rapidamente atingindo um limite físico, imposto simultaneamente pela velocidade da luz no vácuo e pelo tamanho mínimo dos componentes eletrônicos. Ao mesmo tempo, considera-se inaceitavelmente lenta a velocidade de processamento obtida neste modelo computacional em muitos problemas científicos importantes e extremamente complexos, como por exemplo [Lev89]:

- Química quântica, mecânica estatística e física relativista;
- Cosmologia e astrofísica;

- Dinâmica de fluidos e turbulência;
- Desenvolvimento de materiais e supercondutividade;
- Biologia, farmacologia, seqüenciamento de genomas, engenharia genética, síntese de proteínas, atividade enzimática e modelagem de células;
- Medicina e modelagem de ossos e órgãos humanos;
- Climatologia global e modelagem ambiental;
- Reconhecimento de voz em tempo-real;
- Gerenciamento de grandes bases de dados;
- Cripto-análise.

O Paralelismo proporciona uma interessante alternativa para este tradicional paradigma: numa máquina paralela, vários processadores cooperam simultaneamente para resolver um mesmo problema, gastando apenas uma fração do tempo necessário. Evidentemente, esta idéia não é nova: ela tem sido utilizada sempre que surge a necessidade de realizar um trabalho em menor tempo. No entanto, tornou-se viável na computação graças ao declínio do custo e do tamanho dos processadores. Por este motivo, para os interessados em processamento de alto desempenho, o desenvolvimento de algoritmos paralelos é praticamente uma necessidade.

Há uma outra motivação para o uso do Paralelismo, reconhecida mais recentemente e citada com menor freqüência: ele possibilita a resolução de problemas de uma forma mais vantajosa, mesmo quando se dispõe, no modelo seqüencial, de suficiente tempo para aguardar uma resposta. Isto ocorre, por exemplo, em aplicações com as seguintes características: o número de dados ou os seus valores estão em função do tempo; determinados resultados intermediários afetam as entradas futuras; as entradas correspondem a fluxos múltiplos; uma determinada computação já realizada não pode ser revertida. Nestas situações e em outras semelhantes, o que importa já não é mais a velocidade de processamento, mas o que poderíamos chamar de a **habilidade do computador paralelo de estar em mais de um lugar ao mesmo tempo**, graças a seus diversos processadores. É esta peculiaridade do Paralelismo que garante o êxito final de aplicações deste estilo.

Por fim, há ainda uma terceira razão para as pesquisas na área de Paralelismo, desta vez de cunho mais teórico. O uso de vários processadores trabalhando juntos numa mesma

computação representa um outro paradigma na resolução computacional de um problema. Isto significa uma profunda ruptura com o enfoque seqüencial que dominou a Ciência da Computação desde as suas origens. Como é lógico, esta alternativa proporcionou novos resultados teóricos para a maioria dos problemas computacionais, e ofereceu ainda novas técnicas de elaboração e análise de algoritmos. Sem exagero, pode-se dizer que o estudo do Paralelismo trouxe, como sua principal contribuição, um melhor e mais profundo entendimento da própria natureza da Ciência da Computação.

## I.2 Sistema Paralelo *versus* Sistema Distribuído

Apenas com o intuito de esclarecimento, apresentamos concisamente os principais pontos que distinguem os Sistemas Paralelos dos Distribuídos.

Em um Sistema Paralelo, geralmente se procura a resolução mais rápida de uma tarefa através do emprego simultâneo de múltiplos processadores. Segundo esse enfoque, **uma única aplicação é dividida em tarefas que podem ser executadas ao mesmo tempo**.

Por outro lado, um Sistema Distribuído é formado por um conjunto de computadores autônomos interconectados, que cooperam entre si compartilhando recursos. Nele, **as aplicações são divididas em tarefas que podem ser executadas em diferentes locais**.

É fácil ver que ambas as áreas têm muitos pontos em comum: uso de múltiplos processadores, necessidade de comunicação entre eles, vários processos simultâneos que cooperam entre si, etc. No entanto, a ênfase de cada área está em pontos diferentes: em um Sistema Paralelo, além de haver um acoplamento mais forte entre os processadores, procura-se acelerar o processamento de uma única aplicação, utilizando-se memória compartilhada ou redes de interconexão homogêneas, que são conhecidas e exploradas diretamente pelo usuário. Por outro lado, em um Sistema Distribuído, além dos múltiplos recursos estarem fisicamente distantes, há várias aplicações simultâneas de diferentes usuários (a quem se escondem as características internas do sistema), as estruturas são heterogêneas e dinâmicas e nunca se utiliza memória compartilhada.

O quadro a seguir resume bem as diferenças apresentadas:

| SISTEMA PARALELO | SISTEMA DISTRIBUÍDO |
|---|---|
| Múltiplos processadores | Computadores autônomos |
| Tarefas simultâneas | Distribuição de tarefas |
| Acoplamento forte | Recursos fisicamente distantes |
| Procura-se a aceleração de uma única aplicação | Há várias aplicações e usuários |
| Arquiteturas homogêneas | Estruturas heterogêneas e dinâmicas |
| Possibilidade de uso de memória compartilhada | Nunca se utiliza memória compartilhada |

Tendo em conta esta caracterização, vale a pena ressaltar que o trabalho desta tese pertence ao âmbito de um Sistema Paralelo.

## I.3 Modelos de Computação Paralela

O conceito de **modelo científico** tem dois possíveis significados:

- Representação física, matemática ou lógica de uma entidade real, com a finalidade de simular um determinado fenômeno e possibilitar um estudo sistemático de suas propriedades;
- Descrição de um sistema conceitual, na qual se representam as suas principais características.

Os **modelos de computação** servem a ambos os propósitos acima. Primeiramente, são usados para representar entidades reais conhecidas como **computadores**. Neste caso, eles recolhem as características essenciais destas máquinas, ignorando detalhes insignificantes de implementação, e permitem uma descrição abstrata mais simples de se entender e mais fácil de se manipular matematicamente.

Por outro lado, os modelos de computação também são úteis como ferramentas para a elaboração de algoritmos. Neste caso, eles não estão necessariamente ligados a um computador real: seu principal objetivo é possibilitar-nos um entendimento do **processo de computação**, proporcionando uma estrutura para o desenvolvimento de soluções concretas. Dessa maneira, ao se elaborar um método que resolve um problema dentro de um determinado modelo computacional, obtém-se uma significativa descrição desse algoritmo e a possibilidade de se fazer uma análise precisa da sua eficiência.

Considerando mais estritamente a área do Paralelismo, é praticamente impossível falar de algoritmos sem mencionar o **modelo de computação** no qual eles foram projetados. Ao contrário do caso serial, em que a maioria dos computadores pertence a um mesmo modelo, vários modelos foram propostos e usados no estudo do Paralelismo, tanto na teoria como na construção de máquinas. Eles diferem entre si em uma série de pontos,

e a razão óbvia para esta diversidade é que não há um único modelo preferido pelos projetistas. Como conseqüência desta diversidade, surgiu um imenso campo de pesquisa e de projetos.

*Grosso modo*, são duas as principais características que diferenciam os modelos de Computação Paralela: a presença ou não de um sincronismo entre os processadores e o modo como eles se comunicam entre si. Com relação a este último ponto, a comunicação pode ser realizada de duas maneiras: através de uma memória compartilhada ou de uma rede de interconexão. No primeiro caso, é necessário explicitar como são resolvidos os possíveis conflitos de acesso; no segundo, é preciso determinar a estrutura dessa rede.

Apresentaremos a seguir, de uma maneira sucinta, como estes aspectos costumam ser modelados. Primeiramente, trataremos da comunicação entre os processadores, e depois da possível presença de um sincronismo entre eles.

### I.3.1 Comunicação entre os processadores

Como dissemos acima, a comunicação entre os processadores de uma máquina paralela pode ser modelada de duas maneiras: através de uma memória compartilhada (*shared memory*) ou de uma rede de interconexão (*interconnection network*). De modo geral, o primeiro caso é mais eficiente, porém mais caro. É claro que poderíamos pensar também numa rede completa de interconexão, mas sua construção é praticamente inviável quando se tem um elevado número de processadores.

O modelo básico de comunicação através de memória compartilhada permite que todos os processadores tenham acesso a todas as suas posições. No entanto, como eles nem sempre lêem ou escrevem em posições diferentes, é necessário definir quatro modelos de acesso a essa memória:

- **EREW** *(Exclusive Read, Exclusive Write)*: dois processadores não podem ter acesso simultâneo a uma mesma posição de memória.

- **CREW** *(Concurrent Read, Exclusive Write)*: dois ou mais processadores podem ter acesso simultâneo a uma mesma posição da memória apenas para ler, mas não para escrever.

- **CRCW** *(Concurrent Read, Concurrent Write)*: dois ou mais processadores podem ter acesso simultâneo a uma mesma posição da memória para ler ou escrever.

- **ERCW** *(Exclusive Read, Concurrent Write)*: dois ou mais processadores podem ter acesso simultâneo a uma mesma posição da memória apenas para escrever, mas não para ler. Este último modelo foi definido apenas por razões de completude, pois não tem interesse prático.

Permitir leituras simultâneas em uma mesma posição da memória não representa nenhum problema teórico. No entanto, conflitos de escrita exigem alguma política de tratamento, que pode ser, por exemplo: permissão da escrita do processador de menor índice; permissão da escrita apenas quando todos desejam escrever um mesmo valor; armazenamento da soma de todos os valores que estão sendo escritos; etc. Evidentemente, soluções mais sofisticadas implicam um ônus maior, tanto em termos de custo como de tempo.

A outra alternativa de comunicação entre os processadores são as redes de interconexão. Cada par de processadores pode ter ou não um canal particular de comunicação para troca de mensagens: em caso afirmativo, esses processadores são considerados vizinhos entre si.

Na avaliação dessas possíveis topologias, alguns critérios são levados em conta:

- **Grau.** O grau de um processador numa determinada rede é definido como o número de vizinhos que ele possui. Ter um grau alto é interessante do ponto de vista teórico, mas na prática é desejável um grau pequeno, por questões de custo e de viabilidade de construção.

- **Diâmetro.** O diâmetro de uma rede é o comprimento do maior caminho entre dois processadores. É conveniente que esse valor seja pequeno, pois dele depende a estimativa do tempo de comunicação entre os processadores.

- **Comprimento das conexões.** Embora os modelos sejam abstratos, eles podem representar computadores que serão implementados posteriormente. Neste caso, o ideal é que o comprimento das conexões da rede não varie em função do número de processadores.

Há ainda outros critérios importantes na avaliação de uma rede de interconexão e, evidentemente, não é possível atender a todos eles. Por este motivo, são diversas as topologias estudadas na literatura: *array* linear, anel, grade (*mesh*), árvore, hipercubo, embaralhador perfeito (*shuffle exchange*), De Bruijn, estrela, etc. Ao longo deste trabalho, não nos estenderemos mais nesse assunto, pois consideraremos apenas a comunicação através de memória compartilhada.

### I.3.2 Fluxos de instruções e de dados

O funcionamento de qualquer computador, seja seqüencial ou paralelo, consiste na execução de um conjunto de instruções que operam sobre dados de entrada. O algoritmo, que é quem determina o que o computador faz em cada momento, pode ser visto como um fluxo de instruções, enquanto sua entrada pode ser considerada como um fluxo dos dados que elas manipulam.

Quando há um único fluxo de instruções, isto é, quando um único algoritmo dirige a operação de todos os processadores, dizemos que eles trabalham sincronamente. Caso contrário, dizemos que o computador paralelo opera de modo assíncrono.

Segundo a tradicional classificação de Flynn [Fly66], que se baseia no número de fluxos de instruções e de dados, é possível distinguir quatro modelos de Computação Paralela:

- **SISD**: *Single Instruction stream / Single Data stream*

  Há um único processador, que segue apenas um fluxo de instruções e opera também sobre um único fluxo de dados. É o modelo mais comum de computação, conhecido como seqüencial ou *Von Neumann*.

- **MISD**: *Multiple Instruction stream / Single Data stream*

  São vários processadores, cada um com seu próprio controle (isto é, com seu fluxo de instruções), mas que compartilham um único fluxo de dados. Seu paralelismo consiste em que os processadores fazem diferentes coisas simultaneamente sobre esses mesmos dados.

- **SIMD**: *Single Instruction stream / Multiple Data stream*

  São vários processadores operando de uma maneira síncrona (isto é, seguindo um mesmo fluxo de instruções), mas sobre dados diferentes. Estes dados podem estar armazenados na memória local de cada processador ou numa memória compartilhada por todos. Na execução de cada instrução, é possível selecionar um subconjunto destes processadores, que, naquele momento, são chamados de *ativos*. Os *inativos* esperam até a instrução seguinte.

- **MIMD**: *Multiple Instruction stream / Multiple Data stream*

  São vários processadores operando assincronamente sobre diversos fluxos de dados. Esta classe de computadores é a mais ampla e poderosa de todas, pois cada processador tem a capacidade de executar seu próprio programa sobre dados particulares.

Quando a comunicação é realizada através de memória compartilhada, esses computadores são chamados de *multiprocessadores* ou máquinas fortemente acopladas; por outro lado, quando há uma rede de interconexão, eles são conhecidos como *multicomputadores* ou máquinas fracamente acopladas.

Os computadores SIMD são consideravelmente mais versáteis que os MISD: na literatura científica há uma extensa variedade de problemas resolvidos neste modelo. Além disso, seus algoritmos são relativamente mais fáceis de elaborar, analisar e implementar. No entanto, a principal restrição para o modelo SIMD é a exigência de se encontrar sempre subproblemas que sejam idênticos e, obviamente, há muitas computações que não seguem este padrão.

Como dissemos acima, os computadores MIMD são os mais poderosos. No entanto, máquinas assíncronas são mais difíceis de projetar, avaliar e implementar. Seus algoritmos precisam tratar de tarefas ou processos que são executados simultaneamente em um grupo de processadores. Por isso, é fácil perceber a necessidade de uma política de tratamento para esses processos, que considere suas prioridades e a disponibilidade de processadores em cada momento.

Hoje em dia, esta classificação de Flynn tende a ser abandonada, pois se pode dizer — de modo mais simples — que há um modelo de computação seqüencial (o SISD) e dois de Computação Paralela: o síncrono (SIMD) e o assíncrono (MIMD). Ao longo deste trabalho, apresentaremos algoritmos paralelos síncronos, mas que ainda chamaremos de SIMD.

## I.4 Análise de algoritmos paralelos

Dentre as idéias que se propagaram na Ciência da Computação nos últimos 30 anos, o Paralelismo foi uma das principais causas de transformação. Afinal, todos os aspectos da computação foram afetados, dando origem a muitos conceitos novos. Da arquitetura de computadores aos sistemas operacionais, das linguagens de programação e compiladores aos bancos de dados e inteligência artificial, da computação numérica à combinatorial, cada ramo de pesquisa recebeu uma profunda revitalização. Tanto nos círculos teóricos como nos práticos, sentiu-se um forte grau de atividade desconhecido desde os alvoreceres desta ciência.

Provavelmente, o efeito mais forte ocorreu justamente nos fundamentos da computação: na **elaboração e análise de algoritmos**. Quando o tradicional estudo de al-

goritmos já havia atingido um certo grau de maturidade e estabilidade, e os resultados significativos se tornavam raros, esta revolução possibitou um grande rejuvenescimento nesta importante área de pesquisa.

Afinal de contas, embora a arquitetura, o sistema operacional, a linguaguem e o compilador sejam ingredientes necessários na Ciência da Computação, sabemos que o algoritmo é o mais importante, pois sem ele nenhum problema pode ser resolvido. Por este motivo, os algoritmos paralelos correspondem ao **âmago** do Paralelismo: dado um problema a ser resolvido paralelamente, é o algoritmo que determina como será essa resolução em uma determinada máquina, ou seja, como este problema será subdividido, como os processadores se comunicarão e como as soluções parciais serão combinadas para produzir a resposta final.

Em função desta especial relevância, é natural que se estabeleçam alguns critérios de avaliação para os algoritmos paralelos. Os mais importantes são o tempo de execução, o número de processadores utilizados e o número de passos executados. Estes critérios, mais as técnicas empregadas na sua medição e a interpretação dos resultados obtidos, correspondem ao que costumamos chamar de *análise de algoritmos*.

### I.4.1 Notação básica

O cálculo que exprime o comportamento dos algoritmos é grandemente simplificado quando se utilizam determinadas notações. Faremos agora uma breve descrição das notações $\mathcal{O}(.)$, $\Omega(.)$, $\Theta(.)$ e $o(.)$, que são utilizadas com freqüência nestes cálculos.

Seja uma função $f : \mathbb{N} \to \mathbb{N}$. Definimos os seguintes conjuntos de funções:

$$\begin{aligned}
\mathcal{O}(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists\ n_0 \geq 0,\ \exists\ c > 0,\ \text{tal que } \forall\ n \geq n_0,\ g(n) \leq cf(n)\} \\
\Omega(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists\ n_0 \geq 0,\ \exists\ c > 0,\ \text{tal que } \forall\ n \geq n_0,\ g(n) \geq cf(n)\} \\
\Theta(f) &= \mathcal{O}(f) \cap \Omega(f)
\end{aligned}$$

Como é usual, escreveremos $g = \mathcal{O}(f)$, $g = \Omega(f)$ e $g = \Theta(f)$ ao invés de $g \in \mathcal{O}(f)$, $g \in \Omega(f)$ e $g \in \Theta(f)$, respectivamente.

Além disso, seguindo a definição de Wilf [Wil86], dadas duas funções $f(n)$ e $g(n)$, dizemos que:

$$f(n) = o(g(n)) \ \text{ se } \lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0.$$

Portanto, dados dois algoritmos que gastam tempo $f(n)$ e $g(n)$, onde $n$ é o tamanho da entrada, se $f(n) = o(g(n))$ então sabemos que, para valores suficientemente grandes de $n$, o desempenho do primeiro algoritmo será garantidamente superior ao do segundo.

### I.4.2 Passos elementares

Para medirmos o tempo de execução de qualquer algoritmo paralelo, precisamos definir quais são seus **passos elementares**. Eles são de dois tipos:

- **Passos computacionais:** São operações básicas aritméticas ou lógicas executadas por um processador sobre um ou dois dados. De modo geral, requerem sempre um número constante de *unidades de tempo*.
- **Passos de roteamento:** São usados pelos algoritmos para mover um dado de tamanho constante de um processador para outro, através da memória compartilhada ou das ligações que os conectam.

No caso em que os processadores se comunicam através de memória compartilhada, um passo de roteamento significa dois acessos: uma escrita e uma leitura. É comum se supor que cada acesso à memória gasta um número constante de *unidades de tempo*: é a chamada *análise uniforme*, que é muito mais simples de ser feita. **Todos os algoritmos comentados nesta tese seguem este padrão**. Há também a *análise não-uniforme*, segundo a qual cada acesso requer $\mathcal{O}(\log M)$ *unidades de tempo*, onde $M$ é o tamanho da memória. Embora seja mais realista, é pouco utilizada na prática.

Por outro lado, os processadores também podem se comunicar pelo envio de dados ao longo de canais específicos que os unem. Se eles estiverem conectados diretamente, supõe-se que esta comunicação gastará um número constante de *unidades de tempo*. Caso contrário, gastará um tempo proporcional ao tamanho do menor caminho que os une. Por exemplo, para uma máquina paralela de $p$ processadores com topologia *hipercubo*, isto significa tempo $\mathcal{O}(\log p)$ no pior caso.

### I.4.3 Tempo de execução

O *tempo de execução* de um algoritmo paralelo é definido como o tempo que ele exige para resolver o problema em questão. Mais precisamente, é o tempo gasto desde o momento em que o primeiro processador começa a receber os dados de entrada até o instante em que o último processador termina de produzir os dados de saída.

Costuma-se considerar o tempo de execução do pior caso, isto é, o tempo máximo que o algoritmo gasta para resolver um caso específico do problema em questão.

Desde que cada passo elementar (computacional ou de roteamento) gasta um número constante de *unidades de tempo*, o número destes passos é uma boa estimativa teórica da

real quantidade de tempo que o algoritmo necessita em um computador real. O número de passos — e portanto o tempo de execução — de um algoritmo paralelo é uma função do tamanho da entrada e do número de processadores utilizados. Portanto, para um problema de tamanho $n$, o pior caso de tempo de execução de um algoritmo paralelo é indicado por $t(n)$. Por isso, quando dizemos que um algoritmo gasta tempo $t(n)$ ou tem complexidade de tempo $t(n)$, isto significa que $t(n)$ é o número de *unidades de tempo* exigidas por ele na resolução do pior caso do problema.

**Aceleração.** A principal razão para se utilizar algoritmos paralelos é acelerar as computações seqüenciais. Por isso, é muito natural que queiramos comparar o tempo de execução de um algoritmo paralelo elaborado para um determinado problema com o melhor algoritmo seqüencial que também o resolve. Isto costuma ser feito através de uma relação chamada **aceleração** (ou *speedup*), definida como $T_s/T_p$, onde $T_s$ é a complexidade de tempo do melhor algoritmo seqüencial e $T_p$ é a complexidade de tempo do algoritmo paralelo em questão. Esta relação pode ser entendida como quantas vezes se ganhou no tempo através desta paralelização; por isso, quanto maior for a aceleração, melhor será a paralelização. No entanto, o valor máximo que a aceleração pode atingir (salvo algumas anomalias, que não vale a pena comentar) é igual ao número de processadores utilizados; neste caso, pode-se dizer que a paralelização é **ótima**. Se a aceleração fosse maior que este valor, a simulação seqüencial desta paralelização daria origem a um algoritmo seqüencial superior ao melhor conhecido.

### I.4.4 Número de processadores

Um outro critério de medida de desempenho de um algoritmo paralelo é o número de processadores que ele utiliza. Afinal, é evidente que se deseje obter uma boa aceleração com um mínimo possível de processadores.

**Custo.** Para um problema de tamanho $n$, suponha que o número de processadores exigidos por um algoritmo paralelo seja uma função de $n$ denotada por $p(n)$, e que esse algoritmo roda em tempo $t(n)$ no pior caso. Portanto, um limite superior no número total de passos elementares executados por este algoritmo é dado pelo seu **custo** $C(n)$ definido como $p(n) \times t(n)$. Dizemos que é um limite superior porque nem todos os $p(n)$ processadores precisam estar ativos ao longo das $t(n)$ *unidades de tempo*.

**Adaptabilidade.** Uma característica que mede a versatilidade de um algoritmo paralelo é a sua capacidade de resolver a mesma instância de um problema em máquinas com diferentes números de processadores. Quando um algoritmo paralelo tem esta capacidade,

ele é chamado de **adaptativo** (ou **escalável**). Esta característica é desejável porque permite que uma mesma implementação de um algoritmo paralelo possa ser utilizada em máquinas que diferem apenas no número de processadores disponíveis.

## I.5 Âmbito desta tese

Como foi visto ao longo deste capítulo, computadores paralelos podem ser classificados segundo suas características arquiteturais e modos de operação, como por exemplo: tipo de processadores, interconexões entre eles, seus correspondentes esquemas de comunicação, etc. No entanto, é importante ressaltar que tais considerações estão **fora** do âmbito deste trabalho: o nosso objetivo principal é apenas elaborar algoritmos adequados a computadores paralelos. O que se enfatiza é a técnica e o paradigma, ao invés de se detalhar aspectos específicos de determinadas máquinas. Esta postura é bastante comum e seguida por autores renomados [Akl89, JaJ92, Qui94, KGG94, Akl97]; o ponto central da pesquisa é o desenvolvimento e a análise de algoritmos, ficando para um trabalho posterior o estudo de suas possíveis implementações. Este foi também o enfoque do trabalho desenvolvido durante o Mestrado, e posteriormente publicado [SaS94].

De qualquer forma, esta tarefa não é simples, como talvez pareça. A experiência mostra que muitos algoritmos desenvolvidos para computadores seqüenciais não são apropriados para arquiteturas paralelas, ao mesmo tempo que diversos algoritmos com paralelismo inerente têm complexidades piores que o melhor algoritmo seqüencial para o mesmo problema. Portanto, o desenvolvimento e a análise de algoritmos paralelos é um trabalho árduo e empolgante, cujos resultados podem ser de grande interesse prático.

O modelo abordado ao longo desta tese é a conhecida PRAM: *Parallel Random Access Machine*. Ela está representada na figura I.1, e é formada por:

- Um conjunto de $p$ processadores idênticos $P_0, P_1, \ldots, P_{p-1}$. A princípio, o valor de $p$ não é limitado, mas sempre o consideramos como um número finito, embora arbitrariamente grande.

- Uma memória compartilhada com $M$ posições. O valor de $M$ também não é limitado; no entanto, o consideramos arbitrariamente grande, mas finito, onde $M \geq p$.

- Uma unidade de acesso, que permite aos processadores ter acesso à toda memória compartilhada.

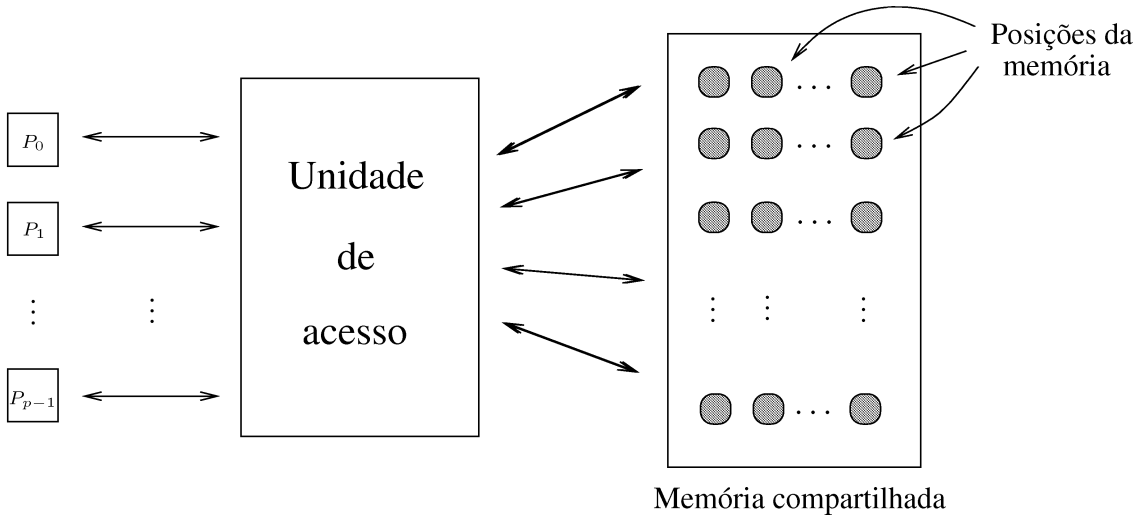


Figura I.1: O modelo PRAM.

Na verdade, nossos resultados foram obtidos para um modelo específico de PRAM, que tem as seguintes características:

- **SIMD**: *Single Instruction/Multiple Data*

  Todos os processadores executam um mesmo código, mas sobre diferentes dados.

- **CREW**: *Concurrent Read/Exclusive Write*

  Uma mesma posição da memória compartilhada pode ser lida simultaneamente por mais de um processador, mas a escrita simultânea não é permitida.

São comuns na literatura trabalhos que consideram a PRAM com uma máquina SIMD (ou seja, síncrona) já na sua própria definição. No entanto, para sermos precisos, não faremos dessa maneira: como pode haver modelos assíncronos de PRAM, iremos especificar esta característica ao longo do texto.

De qualquer maneira, as PRAM's costumam ser consideradas como máquinas teóricas: por questões de inviabilidade prática, não se constróem computadores com as suas características. Por este motivo, nem sempre os resultados obtidos numa PRAM têm uma aplicação direta; no entanto, por se tratar de um modelo poderoso, podemos dizer que acabam sendo mais abrangentes e permanentes. Além disso, vale a pena frisar que, quando se implementam em computadores reais algoritmos que foram elaborados para a PRAM, as complexidades de tempo e de espaço podem ser facilmente recalculadas em função de parâmetros relacionados com as características particulares dessas máquinas.

Muitos autores clássicos da área de Paralelismo — como os citados logo acima —

destacam a PRAM em seus livros, pois é um modelo de grande difusão na literatura científica. Isto também pode ser observado no Capítulo III, quando apresentamos diversas resoluções paralelas do Problema da Mochila. Por isso, a elaboração de algoritmos para a PRAM está intimamente ligada à **Teoria da Computação**, e a obtenção de resultados originais em uma área nobre de pesquisa é um elemento que valoriza este trabalho.

Terminada esta introdução, passaremos agora ao objetivo deste trabalho, que é a elaboração e a análise de algoritmos paralelos mais eficientes para resolução de uma variante do Problema da Mochila na PRAM SIMD CREW. Inicialmente, no Capítulo II apresentaremos as principais variantes do Problema da Mochila e as melhores resoluções seqüenciais conhecidas. Entre outras coisas, se verá que todos os algoritmos seguem apenas dois paradigmas: das **listas** e da **programação dinâmica**. O Capítulo III, como já foi comentado, contém um resumo do *estado-da-arte*: uma descrição concisa dos melhores algoritmos paralelos para a resolução do Problema da Mochila na PRAM SIMD. A contribuição original desta tese encontra-se nos Capítulos IV e V. Ambos estão dedicados a algoritmos paralelos elaborados para a PRAM SIMD CREW: o Capítulo IV baseia-se no paradigma das listas e o Capítulo V no paradigma da programação dinâmica. Por fim, o último capítulo contém uma conclusão do trabalho realizado, com as perspectivas de futuros estudos.

# Capítulo II

# O Problema da Mochila e suas melhores resoluções seqüenciais

Após termos visto as principais características da Computação Paralela, trataremos agora do problema-alvo deste estudo: o conhecido **Problema da Mochila** (*Knapsack Problem*). Mais especificamente, apresentaremos a definição deste problema e os melhores algoritmos seqüenciais desenvolvidos até o momento para a resolução de uma de suas variantes: o *Subset-Sum Problem* (SSP).

Consideraremos apenas os algoritmos que obtêm a **solução exata** do Problema da Mochila, e os compararemos segundo as suas complexidades de tempo e de espaço. Dessa forma, levaremos em conta o desempenho destes algoritmos no **pior caso** do problema, e não o tempo médio em situações práticas.

## II.1 O Problema da Mochila

O **Problema da Mochila**, em todas as suas variantes, é conhecidamente **NP-completo** [GaJ79]: em outras palavras, até hoje não existem algoritmos seqüenciais que sejam capazes de resolvê-lo, no seu pior caso, em tempo polinomial.

Suas características peculiares tornaram-no objeto de intensas pesquisas há quase cinco décadas, desde o famoso trabalho de Dantzig [Dan57]. Ele aparece numa vasta gama de cenários, e sua resolução pode ser vista como um modo de estudar grandes problemas na teoria de números. Por causa da sua complexidade, alguns sistemas criptográficos de chave-pública, como por exemplo o *Merkle-Hellman* [Sti95], baseiam-se nele.

São muitos os esforços realizados para se encontrar técnicas que possibilitem a sua resolução com razoáveis tempos de execução. A elaboração de algoritmos paralelos para ele é um dos possíveis caminhos.

De modo geral, o enunciado deste problema é bastante simples: dados $n$ objetos,

cada um deles com um peso e um lucro determinado, deseja-se acondicionar dentro de uma mochila um grupo desses objetos, de tal forma que o lucro obtido seja máximo e a somatória de seus pesos não ultrapasse a capacidade da mochila. Há alguns aspectos acidentais que podem mudar, dando origem a diversas variantes. Por exemplo: o lucro correspondente a cada objeto pode ser seu próprio peso; no acondicionamento ótimo, pode-se colocar ou não vários objetos iguais; pode haver um limite para o valor dos pesos dos objetos; há uma generalização para múltiplas mochilas; etc.

Ao longo deste texto — apenas com a exceção de dois algoritmos[1] —, chamaremos de $V = \{v_0, v_1, \cdots, v_{n-1}\}$ o conjunto desses $n$ objetos, onde cada um deles terá peso $w_i$ e lucro $l_i$, com $w_i$ e $l_i$ inteiros positivos, $0 \leq i < n$. Também chamaremos de $W$ a coleção dos $n$ pesos, onde $w_{min}$ e $w_{max}$ são, respectivamente, o menor e o maior valor em $W$. Considerando a capacidade $c$ da mochila como um número inteiro positivo, suporemos sem perda de generalidade que $\sum_{i=0}^{n-1} w_i > c > w_{max}$.

### II.1.1 Algumas de suas variantes

Antes de abordarmos o SSP (*Subset-Sum Problem*), que é a variante que nos interessa mais diretamente, vamos fazer uma breve apresentação de outros problemas dessa mesma família.

Outra variante que também nos interessa é o **Problema da Mochila 0-1** (*0-1 Knapsack Problem*): consiste em escolher um subconjunto dos $n$ objetos de tal forma que se maximize o lucro correspondente, sem que a capacidade da mochila seja excedida. Este problema, que será chamado de KP01, pode ser formulado do seguinte modo:

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} l_i x_i \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_i x_i \leq c \\ & x_i \in \{0,1\}, 0 \leq i < n \end{aligned}$$

Como se deseja encontrar um subconjunto dos objetos, as variáveis $x_i$ são binárias: daí o nome dessa variante. Quando $x_i = 1$, o objeto $v_i$ é incluído na mochila; caso contrário, ele é excluído. A solução deste problema pode ser representada por um vetor binário $X = \{x_0, x_1, \ldots, x_{n-1}\}$, onde $x_i \in \{0,1\}$, $0 \leq i < n$.

[1] Na apresentação dos algoritmos de Pisinger (neste mesmo capítulo) e de Ferreira e Robson (no próximo capítulo), seguiremos a notação desses próprios autores, que consideram os índices $i$ dos objetos, pesos e lucros variando entre 1 e $n$, ao invés do intervalo entre 0 e $n-1$.

Quando houver uma quantidade limitada $m_i$ para cada objeto $v_i$, temos o **Problema Limitado da Mochila** (*Bounded Knapsack Problem*):

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} l_i x_i \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_i x_i \leq c \\ & x_i \in \{0, 1, \ldots, m_i\}, 0 \leq i < n \end{aligned}$$

A generalização deste problema corresponde ao **Problema Ilimitado da Mochila** (*Unbounded Knapsack Problem*), onde se tem uma ilimitada disponibilidade de cada objeto. Ao longo desta tese, ele será chamado de KP:

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} l_i x_i \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_i x_i \leq c \\ & x_i \geq 0 \text{ inteiro}, 0 \leq i < n \end{aligned}$$

Neste caso, a solução do problema também pode ser representada por um vetor $X = \{x_0, x_1, \ldots, x_{n-1}\}$ que, no entanto, deixa de ser binário.

Se desejamos escolher alguns dos $n$ objetos para empacotá-los em $m$ mochilas diferentes de capacidades $c_j$, de tal forma que a maior soma dos lucros seja obtida, teremos então o **Problema das Múltiplas Mochilas** (*Multiple Knapsack Problem*):

$$\begin{aligned} \text{maximizar} \quad & \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} l_i x_{ij} \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_i x_{ij} \leq c_j, \quad 0 \leq j < m \\ & \sum_{j=0}^{m-1} x_{ij} \leq 1, \qquad 0 \leq j < m \\ & x_{ij} \in \{0, 1\}, \qquad 0 \leq i < n,\ 0 \leq j < m \end{aligned}$$

A restrição $\sum_{j=0}^{m-1} x_{ij} \leq 1$ garante que cada objeto seja escolhido no máximo uma única vez.

Há ainda uma formulação mais geral do Problema da Mochila, que é conhecida como **Problema da Mochila Multi-restrito** (*Multi-constrained Knapsack Problem*), na qual todos os coeficientes $w_{ij}$, $l_i$ e $c_j$ são inteiros não negativos:

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} l_i x_i, \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_{ij} x_i \leq c_j, \quad 0 \leq j < m \\ & x_i \geq 0 \text{ inteiro}, \quad 0 \leq i < n \end{aligned}$$

Ainda é possível encontrar na literatura outras variantes pertencentes à família do Problema da Mochila (*Multiple-choice Knapsack Problem*, *Change-making Problem*, *Bin-packing Problem*, *Collapsing Knapsack Problem*, *Nested Knapsack Problem*, *Nonlinear Knapsack Problem*, *Inverse-parametric Knapsack Problem*), mas que não serão objeto da nossa pesquisa.

### II.1.2 *Subset-Sum Problem*

As principais contribuições desta tese referem-se a uma variante bastante simples e conhecida, que é um caso particular do KP01 onde o lucro $l_i$ é igual ao peso $w_i$, para todos os $n$ objetos. Chama-se *Subset-Sum Problem* (SSP):

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=0}^{n-1} w_i x_i \\ \text{sujeito a} \quad & \sum_{i=0}^{n-1} w_i x_i \leq c \\ & x_i \in \{0,1\}, 0 \leq i < n \end{aligned}$$

Como o próprio nome diz, este problema pode ser visto como a escolha de um subconjunto dos pesos de tal forma que sua soma seja máxima, sem que se exceda a capacidade da mochila. Da mesma forma que o KP01, sua solução também pode ser representada por um vetor binário $X = \{x_0, x_1, \ldots, x_{n-1}\}$, onde $x_i \in \{0,1\}$, $0 \leq i < n$.

Alguns dos algoritmos que encontramos na literatura — e que serão apresentados mais adiante — resolvem um caso particular do SSP, que chamaremos de *Subset-Sum Decision Problem* ou SSPd . Neste problema, deseja-se apenas saber se existe um acondicionamento dos $n$ objetos que atinge a capacidade $c$ da mochila:

$$\begin{aligned} & \sum_{i=0}^{n-1} w_i x_i = c \\ & x_i \in \{0,1\}, 0 \leq i < n \end{aligned}$$

## II.2 Algoritmos seqüenciais segundo o paradigma das listas

A literatura científica consagrou dois paradigmas para a resolução exata do Problema da Mochila: o **paradigma das listas** e o **paradigma da programação dinâmica**.

O primeiro surgiu com Horowitz e Sahni [HoS74], quando resolveram o KP01 através do uso de duas listas de subsoluções. O resultado obtido por este algoritmo corresponde até hoje ao *upper bound* em termos de tempo e espaço na resolução do Problema da Mochila, quando esses valores são expressos unicamente em função do número $n$ de objetos. Posteriormente, Schroeppel e Shamir [ScS81] mostraram que o uso de quatro listas ao invés de duas pode reduzir o espaço necessário na resolução do SSPd, embora acrescente um fator extra no tempo. Ao mesmo tempo, provaram que nenhum outra modificação no número de listas é capaz de melhorar estes resultados. Embora Schroeppel e Shamir não comentem, este algoritmo pode ser adaptado para a resolução do SSP e do KP01, sem alterações nas suas complexidades. É por este motivo que muitos dos algoritmos paralelos desenvolvidos posteriormente são dedicados apenas à resolução do SSPd, conforme se pode verificar no próximo capítulo.

O outro paradigma deve-se a Bellman [Bel57], responsável pelo conhecido método chamado de **programação dinâmica**, útil na resolução de uma grande variedade de problemas. Concretamente, esta técnica também pode ser aplicada ao KP, obtendo um bom desempenho na maioria dos casos práticos. No entanto, sua complexidade de tempo e de espaço deixa de depender apenas do número $n$ de objetos, mas também da capacidade $c$ da mochila.

Vale a pena mencionar que os algoritmos apresentados pelos autores acima [HoS74, ScS81, Bel57] encontram o lucro ótimo da mochila, mas não recuperam o correspondente vetor $X$. No caso do algoritmo de Bellman, este vetor pode ser calculado facilmente sem qualquer ônus no tempo ou no espaço, mas o mesmo não ocorre nos outros dois algoritmos, que exigem um fator extra $n$ no espaço necessário. Comentaremos isso oportunamente, pois vamos apresentar cada algoritmo individualmente.

### II.2.1 Algoritmo de Horowitz e Sahni

Horowitz e Sahni [HoS74] apresentaram um algoritmo seqüencial que resolve o KP01 em tempo e espaço $\mathcal{O}(2^{n/2})$ e que se tornou conhecido como o *algoritmo das duas listas*. Iremos descrever apenas uma versão simplificada que resolve o SSP; de qualquer forma,

as complexidades são as mesmas para ambos os problemas.

Este algoritmo pode ser visto como uma aplicação do conhecido paradigma da **divisão e conquista**, e por isso tem duas fases distintas:

(i) *Fase de geração:* Corresponde à construção de duas listas ordenadas: cada uma delas contém todas as possíveis soluções do subproblema obtido a partir da metade dos $n$ objetos.

(ii) *Fase de busca:* Encontra a possível solução ótima através de uma combinação entre elementos das duas listas.

Para simplificar a notação na descrição deste algoritmo, cada lista será representada por um vetor unidimensional, cujo primeiro elemento tem índice 0.

**Fase de geração:**

1. Divida $W$ em duas partes $W_1$ e $W_2$, ambas de tamanho $n/2$;

2. Crie uma lista $A$ que contém, em ordem não-decrescente, todas as $2^{n/2}$ possíveis somas de elementos de $W_1$;

3. Crie uma lista $B$ que contém, em ordem não-crescente, todas as $2^{n/2}$ possíveis somas de elementos de $W_2$.

As listas $A$ e $B$ podem ser geradas recursivamente (*cfr.* [Fer91]), como descreveremos a seguir. Suponha, sem perda de generalidade, que $w_0$ e $w_1$ pertencem a $W_1$, e que $w_0 \leq w_1$. Inicialmente, a lista $A$ será $\{0, w_0\}$, ou seja, $A[0] = 0$ e $A[1] = w_0$. Constrói-se então outra lista somando-se $w_1$ a cada elemento da primeira lista, obtendo-se $\{w_1, w_0 + w_1\}$. A intercalação dessas duas listas ordenadas — que será a nova lista $A$ — pode ser feita em tempo linear no número de elementos, e contém todas as possíveis somas de $\{w_0, w_1\}$ em ordem não-decrescente. Basta, portanto, repetir o processo para todos os outros pesos de $W_1$. Deste modo, a *fase de geração* pode ser executada em tempo $\mathcal{O}(2^{n/2})$, uma vez que a criação da lista $B$ é feita de modo análogo.

Depois da geração de ambas as listas $A$ e $B$, sabemos que $A[0] = B[2^{n/2} - 1] = 0$. Dessa forma, é possível realizar a *fase de busca*, que retorna um vetor *opt* tri-dimensional, onde $opt[0]$ é o valor da solução ótima, e $opt[1] = i_{opt}$ e $opt[2] = j_{opt}$ são seus respectivos índices nas listas $A$ e $B$.

**Fase de busca:**

$i \leftarrow 0$

$j \leftarrow 0$

$fim \leftarrow$ **false**

$opt \leftarrow [0, 0, 0]$

**while** $(i < 2^{n/2})$ **and** $(j < 2^{n/2})$ **and** (**not** $fim$) **do**

$y \leftarrow A[i] + B[j]$

**if** $y = c$ **then**

$opt \leftarrow [c, i, j]$

$fim \leftarrow$ **true**

**else if** $y > c$ **then** $j \leftarrow j + 1$

**else**

**if** $y > opt[0]$ **then** $opt \leftarrow [y, i, j]$ **end if**

$i \leftarrow i + 1$

**end if**

**end if**

**end while**

**return** $(opt)$

Como esta fase gasta claramente tempo $\mathcal{O}(2^{n/2})$, o *algoritmo das duas listas* tem complexidade final $\mathcal{O}(2^{n/2})$, tanto de tempo como de espaço.

Até o presente momento, este é o melhor resultado conhecido — em termos de complexidade expressa em função do número $n$ de objetos — para a solução seqüencial do SSP (e também do KP01). É um problema aberto se $\mathcal{O}(2^{n/2})$ é um *lower bound* de tempo para a resolução seqüencial deste problema.

Os autores não mencionam, mas caso se deseje encontrar o vetor $X$, é preciso que, junto de cada elemento das listas $A$ e $B$, seja criado durante a *fase de geração* um vetor binário de $n/2$ posições, que indica os pesos utilizados em cada uma das somatórias. Dessa forma, através dos índices $i_{opt}$ e $j_{opt}$, obtém-se dois vetores binários que formam o vetor $X$. Embora isto não cause aumento na complexidade de tempo, acarreta um gasto extra de espaço, que passa a ser $\mathcal{O}(n2^{n/2})$.

### II.2.2 Algoritmo de Schroeppel e Shamir

Através de uma variação do *algoritmo das duas listas*, Schroeppel e Shamir [ScS81] reduziram as exigências de memória: mostraram como é possível resolver o SSPd em tempo

$\mathcal{O}(n2^{n/2})$ exigindo somente espaço $\mathcal{O}(2^{n/4})$. Este trabalho ficou posteriormente conhecido como o *algoritmo das quatro listas*.

Vamos apresentar aqui uma versão deste algoritmo que resolve o SSPd, mas esse resultado pode ser generalizado para o SSP e o KP01. Este algorimo também tem as mesmas duas fases: a de geração e a de busca. No entanto, os elementos das listas $A$ e $B$ são gerados à medida em que são solicitados pela *fase de busca*. Para que isso seja possível, duas listas $L_1$ e $L_2$ e um *heap* $H_1$ simulam a lista $A$, e outras duas listas $L_3$ e $L_4$ e outro *heap* $H_2$ simulam a lista $B$.

**Fase de geração:**

1) Divida $W$ em quatro partes $W_1, W_2, W_3$ e $W_4$ de mesmo tamanho $n/4$.

2) Crie quatro listas $L_1, L_2, L_3$ e $L_4$ que contêm, respectivamente, todas as possíveis somas de elementos de $W_1, W_2, W_3$ e $W_4$. A lista $L_2$ deve estar em ordem crescente, e a lista $L_4$, em ordem decrescente. Chamaremos de $L_j[i]$ o $i$-ésimo elemento da lista $L_j$, onde $0 \leq i < 2^{n/4}$ e $j = 1, 2, 3, 4$.

3) Crie uma estrutura de *min-heap* $H_1$ formada por todos os pares $(L_1[i], L_2[1])$, para $0 \leq i < 2^{n/4}$, que retorna o par cuja soma de seus elementos seja a **menor**.

4) De modo semelhante, crie uma outra estrutura de *max-heap* $H_2$ formada por todos os pares $(L_3[i], L_4[1])$, para $0 \leq i < 2^{n/4}$, que retorna o par cuja soma de seus elementos seja a **maior**.

De modo análogo à geração das listas $A$ e $B$ do *algoritmo das duas listas*, as quatro listas $L_1, L_2, L_3$ e $L_4$ podem ser criadas em tempo $\mathcal{O}(2^{n/4})$.

A criação de ambos os *heaps*, ou seja, a inserção dos correspondentes pares nessas estruturas, gasta tempo $\mathcal{O}(n2^{n/4})$. $H_1$ e $H_2$ nunca terão mais do que $2^{n/4}$ pares cada um, e o uso destas duas estruturas torna desnecessária a ordenação das listas $L_1$ e $L_3$, como se pode ver na fase seguinte.

**Fase de busca:**

$opt \leftarrow 0$

**while** $H_1$ e $H_2$ não estejam ambas vazias **do**

Seja $(L_1[i], L_2[j])$ o par com menor soma em $H_1$

Seja $(L_3[k], L_4[l])$ o par com maior soma em $H_2$

$S \leftarrow L_1[i] + L_2[j] + L_3[k] + L_4[l]$

**if** $S = c$ **then**
$opt \leftarrow c$
**HALT**
**else if** $S < c$ **then**
**if** $opt < S$ **then** $opt \leftarrow S$ **end if**
Elimine $(L_1[i], L_2[j])$ de $H_1$
**if** $j < 2^{n/4} - 1$ **then** insira $(L_1[i], L_2[j+1])$ em $H_1$ **end if**
**else**
Elimine $(L_3[k], L_4[l])$ de $H_2$
**if** $l < 2^{n/4} - 1$ **then** insira$(L_3[k], L_4[l+1])$ em $H_2$ **end if**
**end if**
**end if**
**end while**
**return**($opt$)

A inserção de um novo elemento num *heap* gasta tempo logarítmico com relação ao número de elementos presentes nessa estrutura, enquanto que a obtenção do menor valor do *heap* (ou maior, conforme sua organização interna) é feita em tempo constante. Dessa forma, a *fase de busca* deste algoritmo gasta tempo $\mathcal{O}(n2^{n/2})$.

Portanto, o *algoritmo das quatro listas* gasta tempo total $\mathcal{O}(n2^{n/2})$. Seu espaço é $\mathcal{O}(2^{n/4})$, que é o tamanho das quatro listas e dos dois *heaps*. É importante ressaltar que os autores deste algoritmo, na análise da complexidade de tempo, desprezaram o fator $n$ correspondente ao uso dos *heaps*. Isso pode ser observado inclusive no título do artigo publicado [ScS81].

Do mesmo modo que no artigo do *algoritmo das duas listas*, não há qualquer comentário sobre a recuperação do vetor $X$ correspondente à solução ótima, que também ocasionaria um fator extra $n$ na complexidade de espaço.

## II.3 Algoritmos seqüenciais segundo o paradigma da programação dinâmica

Passaremos agora a abordar a resolução do Problema da Mochila a partir do paradigma da programação dinâmica. Esta mudança de paradigma pode ser percebida claramente através das complexidades dos algoritmos, pois suas expressões deixam de ser funções exclusivas do número $n$ de objetos. Nestes cálculos, surgem outras variáveis: na maioria

das vezes, será a capacidade $c$ da mochila, mas também podem ser valores relativos aos pesos dos objetos, como $w_{max}$ e $w_{min}$.

Este fato dificulta muito a comparação teórica dos algoritmos de ambos os paradigmas: esta é a principal razão que nos leva a apresentá-los separadamente. Além disso, torna ainda mais complicado o estabelecimento de um *lower bound* para a resolução seqüencial do Problema da Mochila. Apenas a título de exemplo, vale a pena comentar que há casos particulares deste problema que podem ser resolvidos em tempo $\mathcal{O}(nw_{min})$ [Gre80], que normalmente é muito inferior ao tempo obtido pelo tradicional algoritmo de Bellman.

### II.3.1 Algoritmo de Bellman

Bellman [Bel57] foi o criador do paradigma da programação dinâmica, e o primeiro a utilizá-lo na resolução do Problema da Mochila. Como dissemos, esta técnica pode ser aplicada ao KP, mas, restringindo-se ao âmbito do nosso trabalho, apresentaremos apenas uma versão que resolve o KP01.

Chamando o problema KP01 de KNAP$(V, c)$, onde $V$ é o conjunto dos $n$ objetos e $c$ a capacidade da mochila, seja KNAP$(V_i, j)$ o subproblema correspondente a uma mochila de capacidade $j$ considerando apenas os primeiros $i+1$ objetos, ou seja, $V_i = \{v_0, v_1, \ldots, v_i\}$ e $j \leq c$. Seja também $f_i(j)$ o valor da solução ótima para KNAP$(V_i, j)$, isto é, o lucro máximo para esse subproblema.

É fácil perceber que se pode chegar a uma solução ótima para KNAP$(V_i, j)$ a partir das soluções ótimas de subproblemas menores. Isso está descrito no seguinte princípio de otimalidade:

$$\begin{aligned} f_{-1}(j) &= \begin{cases} -\infty, & \text{se } j < 0 \\ 0, & \text{se } j \geq 0 \end{cases} \\ f_i(j) &= \max\{f_{i-1}(j), f_{i-1}(j - w_i) + l_i\},\ 0 \leq j \leq c,\ 0 \leq i < n \end{aligned}$$

Seja $F_i(c) = (f_i(0), f_i(1), \ldots, f_i(c))$ o vetor de lucro ótimo para KNAP$(V_i, c)$, onde $0 \leq i < n$. Dessa forma, KP01 pode ser resolvido por programação dinâmica calculando-se sucessivamente $F_0(c)$, $F_1(c)$, ..., $F_{n-1}(c)$.

O algoritmo abaixo, que tem complexidade de tempo e de espaço $\mathcal{O}(nc)$, realiza estes cálculos:

**for** $j \leftarrow -w_{max}$ **to** $-1$ **do**
    $f_{-1}[j] \leftarrow -\infty$
**end for**
**for** $j \leftarrow 0$ **to** $c$ **do**

$f_{-1}[j] \leftarrow 0$
**end for**
**for** $i \leftarrow 0$ **to** $n-1$ **do**
  **for** $j \leftarrow 0$ **to** $w_i - 1$ **do**
    $f_i[j] \leftarrow f_{i-1}[j]$
  **end for**
  **for** $j \leftarrow w_i$ **to** $c$ **do**
    $f_i[j] \leftarrow \max\{f_{i-1}[j], f_{i-1}[j-w_i] + l_i\}$
  **end for**
**end for**

Caso se deseje, o vetor $X$ pode ser encontrado em tempo $\mathcal{O}(n)$ através de um percurso nos valores de $f$, ou seja, sem alterar as complexidades:

$r \leftarrow c$
$s \leftarrow f_{n-1}[c]$
**for** $i \leftarrow n-1$ **downto** $0$ **do**
  **if** $f_{i-1}[r] = s$ **then** $x_i \leftarrow 0$
    **else**
      $x_i \leftarrow 1$
      $s \leftarrow s - l_i$
      $r \leftarrow r - w_i$
  **end if**
**end for**

Costuma-se dizer que $\mathcal{O}(nc)$ é uma complexidade **pseudopolinomial**, pois é polinomial no tamanho $c$ da mochila, mas não necessariamente no tamanho $n$ da entrada: quando $c = \Theta(2^n)$, uma mochila de tamanho exponencial pode ser especificada por um número polinomial de *bits*. Como $nc < 2^n$ na maioria dos casos práticos, a programação dinâmica tem sido um método eficiente para a resolução do Problema da Mochila.

### II.3.2 Algoritmo de Yanasse e Soma

Yanasse e Soma [YaS87] elaboraram um algoritmo seqüencial para o SSP que é uma variação da tradicional aplicação do paradigma da programação dinâmica: utiliza apenas um único vetor de $c+1$ posições, que aqui será chamado de $g$. Este vetor é calculado

de tal forma que, se houver um preenchimento da mochila de capacidade $k$ tal que o peso de maior índice neste preenchimento seja $w_i$, então $g[k] = i$. Caso haja mais de um preenchimento com a mesma capacidade, $g$ armazenará o menor índice. Além disso, $g[k] = n$ significa a não existência de nenhum preenchimento que atinja esta capacidade $k$.

O algoritmo que calcula o vetor $g$ e a solução *opt* do SSP está logo abaixo, e uma de suas iterações está esquematizada na figura II.1.

**for** $i \leftarrow 0$ **to** $c$ **do**
    $g[i] \leftarrow n$
**end for**
**for** $i \leftarrow n-1$ **downto** $0$ **do**
    $g[w_i] \leftarrow i$
**end for**
$opt \leftarrow w_{max}$
**for** $i \leftarrow w_{min}$ **to** $c - w_{min}$ **do**
    **for** $j \leftarrow g[i]+1$ **to** $n-1$ **do**
        $w' \leftarrow i + w_j$
        **if** $w' \leq c$ **then**
            $g[w'] \leftarrow \min\{g[w'], j\}$
            $opt \leftarrow \max\{opt, w'\}$
        **end if**
    **end for**
**end for**

O próximo algoritmo calcula o vetor binário $X$ em tempo $\mathcal{O}(n)$:

**for** $i \leftarrow 0$ **to** $n-1$ **do**
    $x_i \leftarrow 0$
**end for**
$i \leftarrow opt$
**while** $i \neq 0$ **do**
    $x_{g[i]} \leftarrow 1$
    $i \leftarrow i - w_{g[i]}$
**end while**

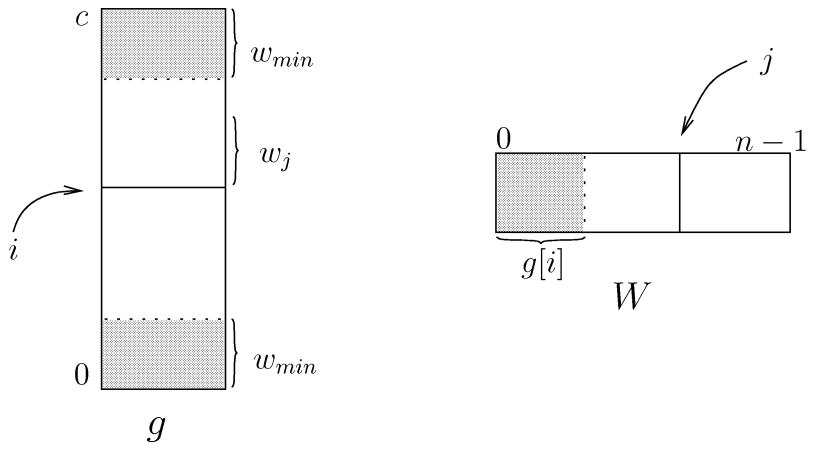


Figura II.1: Cálculo do vetor $g$ no algoritmo de Yanasse e Soma.

Além de ser muito simples, o algoritmo de Yanasse e Soma é mais eficiente que o algoritmo de Bellman, pois resolve o SSP em tempo $\mathcal{O}(n(c-2w_{min})+c)$, gastando apenas espaço $\mathcal{O}(n+c)$.

### II.3.3 Algoritmo de Pisinger

Pisinger [Pis99] introduziu a técnica do **balanceamento** para a resolução de diversas variantes do Problema da Mochila. Tendo em conta o objetivo do nosso trabalho, apresentaremos apenas o algoritmo balanceado para o SSP[2].

Quando os objetos são inseridos sucessivamente na mochila, sem considerar nenhuma ordenação prévia, chamamos de objeto de quebra $b$ o primeiro a não caber na mochila. Dessa forma, $b = \min\{j : \sum_{i=1}^{j} w_i > c\}$. A solução (ou preenchimento) de quebra $x'$ é aquela obtida pela inclusão dos primeiros $b$ objetos, isto é, $x'_j = 1$ para $j = 1, \ldots, b-1$ e $x'_j = 0$ para $j = b, \ldots, n$. O valor correspondente à solução de quebra é $\bar{w} = \sum_{j=1}^{b-1} w_j$.

Considerando a solução de quebra como o primeiro preenchimento balanceado da mochila, novas soluções também balanceadas podem ser obtidas através de duas operações:

- **Inserção balanceada:** Se num preenchimento balanceado $x$ com $\sum_{j=1}^{n} w_j x_j \leq c$ mudamos uma variável $x_t$ $(t \geq b)$ de 0 para 1, então o novo preenchimento obtido também será balanceado.

- **Remoção balanceada:** Se num preenchimento balanceado $x$ com $\sum_{j=1}^{n} w_j x_j > c$ mudamos uma variável $x_s$ $(s < b)$ de 1 para 0, então o novo preenchimento obtido também será balanceado.

[2]Como já foi comentado, seguiremos a notação original na descrição deste algoritmo, ou seja, os índices dos objetos, pesos e lucros variam entre 1 e $n$.

Em outras palavras: numa inserção balanceada a capacidade máxima da mochila ainda não havia sido atingida, e por isso pode entrar o objeto $a_t$, onde $t \geq b$; analogamente, numa remoção balanceada a capacidade máxima da mochila já havia sido ultrapassada, e por isso sai o objeto $a_s$, onde $s < b$.

É fácil perceber que o valor de qualquer preenchimento balanceado está sujeito ao intervalo $c - w_{max} < \sum_{j=1}^{n} w_j x_j \leq c + w_{max}$. Pisinger demonstrou que uma solução ótima é também um preenchimento balanceado, isto é, pode ser obtida através das duas operações descritas acima; mais ainda: esta solução pode ser obtida através de operações balanceadas que consideram os índices $\{t_i\}$ em ordem crescente e os índices $\{s_i\}$ em ordem decrescente, o que sugere um algoritmo.

Antes de descrevermos este algoritmo, são necessárias algumas definições. Seja $f_{s,t}(\tilde{c})$ uma solução ótima para o subproblema de SSP definido sobre os objetos $\{a_s, \ldots, a_t\}$, onde $s \leq b$, $t \geq b-1$, $c - w_{max} < \tilde{c} \leq c + w_{max}$:

$$f_{s,t}(\tilde{c}) = \max \left\{ \begin{array}{l} \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j : \\ \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j \leq \tilde{c}, \\ x_j \in \{0,1\} \text{ para } j = s, \ldots, t, \\ x \text{ é um preenchimento balanceado} \end{array} \right\}$$

onde definimos $\sum_{j=s}^{t} w_j = 0$ se $s > t$.

No algoritmo, somente serão considerados os estados $(s, t, \mu)$, onde $\mu = f_{s,t}(\mu)$, ou seja, soluções de valor $\mu$ obtidas pela aplicação de operações balanceadas nas variáveis $x_s, \ldots, x_t$. Além disso, se houver dois estados $(s, t, \mu)$ e $(s', t', \mu)$ correspondentes a soluções de mesmo valor, será considerado somente aquele que estiver mais **próximo** da solução de quebra, isto é, o que for alcançado com um menor número de inserções e remoções balanceadas. Portanto, se $s \geq s'$ e $t \leq t'$, então o estado $(s', t', \mu)$ é descartado.

Seguindo esta regra, o algoritmo enumera os estados variando $t$ de $b-1$ até $n$. Dessa forma, em cada estágio $t$ e para cada valor de $\mu$ teremos apenas um único índice $s$, que é o maior $s$ tal que uma solução com valor $\mu$ pode ser obtida através de operações balanceadas nas variáveis $x_s, \ldots, x_t$.

Considerando $t = b-1, \ldots, n$ e $c - w_{max} < \mu \leq c + w_{max}$, este índice $s_t(\mu)$ é definido por:

$$s_t(\mu) = \max s \left\{ \begin{array}{l} \text{existe um preenchimento balanceado } x \text{ que satisfaz} \\ \sum_{j=1}^{s-1} w_j + \sum_{j=s}^{t} w_j x_j = \mu \\ x_j \in \{0,1\} \text{ para } j = s, \ldots, t \end{array} \right.$$

No início do algoritmo, um único valor de $s$ é positivo: $s_{b-1}(\bar{w}) = b$, que corresponde à solução de quebra. Os demais elementos recebem valor 0, significando a não existência de

nenhum preenchimento balanceado. Uma solução ótima para SSP é encontrada no último estágio calculando-se $z = \max\{\mu \leq c : s_n(\mu) \geq 0\}$.

**for** $\mu \leftarrow c - w_{max} + 1$ **to** $c$ **do** {*inicialização*}
    $s_{b-1}[\mu] \leftarrow 0$
**end for**
**for** $\mu \leftarrow c + 1$ **to** $c + w_{max}$ **do**
    $s_{b-1}[\mu] \leftarrow 1$
**end for**
$s_{b-1}[\bar{w}] \leftarrow b$
**for** $t \leftarrow b$ **to** $n$ **do**
    **for** $\mu \leftarrow c - w_{max} + 1$ **to** $c + w_{max}$ **do** {$w_t$ *não é inserido*}
        $s_t[\mu] \leftarrow s_{t-1}[\mu]$
    **end for**
    **for** $\mu \leftarrow c - w_{max} + 1$ **to** $c$ **do** {$w_t$ *é inserido*}
        $\mu' \leftarrow \mu + w_t$
        $s_t[\mu'] \leftarrow \max\{s_t[\mu'], s_{t-1}[\mu]\}$
    **end for**
    **for** $\mu \leftarrow c + w_t$ **downto** $c + 1$ **do** {*remoção de um ou vários objetos*}
        **for** $j \leftarrow s_t[\mu] - 1$ **downto** $s_{t-1}[\mu]$ **do**
            $\mu' \leftarrow \mu - w_j$
            $s_t[\mu'] \leftarrow \max\{s_t[\mu'], j\}$
        **end for**
    **end for**
**end for**

Os dois primeiros *loops* gastam tempo $\mathcal{O}(w_{max})$, enquanto as cópias de $s_{t-1}$ em $s_t$ (quando $w_t$ não é inserido) e os cálculos de $s_t$ (quando $w_t$ é inserido) gastam tempo $\mathcal{O}(nw_{max})$.

Para calcularmos o tempo correspondente à remoção de um ou vários objetos, é necessário reparar que, para cada $\mu > c$, os cálculos de $s_t$ são executados $s_n(\mu) \leq b$ vezes. Portanto, durante todo o processo, estes mesmos cálculos serão executados no máximo $bw_{max}$ vezes, consumindo tempo $\mathcal{O}(nw_{max})$.

Com isso, o tempo total gasto pelo algoritmo é $\mathcal{O}(nw_{max})$. Esta também é a ordem do espaço necessário para armazenar todos os vetores $s$.

Pisinger não comenta nada a respeito da recuperação do vetor $X$ a partir dos vetores $s_t$. Ao que parece, isto exigiria modificações no algoritmo que alterariam as complexidades obtidas.

### II.3.4 Algoritmo de Soma e Toth

O algoritmo para o SSP que Soma e Toth [SoT02] conceberam é uma espécie de "resolução mista", pois aplica ambos os paradigmas. No que se refere à programação dinâmica, ele se baseia no algoritmo de Yanasse e Soma [YaS87] e, resumidamente, é composto por quatro fases principais:

*(i)* Dentre os $n$ objetos, separe um subconjunto de $\log_2 c$ objetos e calcule por enumeração exaustiva todas as $2^{\log_2 c} = c$ possíveis combinações, armazenando-as no vetor $g_1$ de modo análogo ao algoritmo de Yanasse e Soma, ou seja, $g_1[k] = i$ significa que há um preenchimento da mochila com capacidade $k$ tal que o peso de maior índice neste preenchimento é $w_i$. Caso haja mais de um preenchimento com a mesma capacidade, $g_1$ armazenará o menor índice.

*(ii)* Para mais outros $\log_2 c$ objetos, monte um segundo vetor $g_2$ de forma análoga ao passo anterior.

*(iii)* Aplique o algoritmo de Yanasse e Soma nos $n - 2\log_2 c$ objetos restantes, utilizando o mesmo vetor $g_1$ já montado no passo *(i)*.

*(iv)* De modo semelhante à *fase de busca* do *algoritmo das duas listas*, percorra os vetores $g_1$ e $g_2$, um em ordem crescente e outro em ordem decrescente, para assim se encontrar a solução final.

É fácil observar que as fases *(i)*, *(ii)* e *(iv)* gastam tempo $\mathcal{O}(c)$, enquanto a fase *(iii)* consome tempo $\mathcal{O}((n - 2\log_2 c)(c - 2w_{min}))$. Portanto, a complexidade total de tempo deste algoritmo é $\mathcal{O}((n - \log_2 c^2)(c - 2w_{min}) + c)$. O espaço gasto continua limitado em $\mathcal{O}(n + c)$, que é o utilizado pelo algoritmo de Yanasse e Soma, o que também permite a recuperação do vetor $X$ em tempo $\mathcal{O}(n)$, ou seja, sem modificar a complexidade.

## II.4 Quadro comparativo dos algoritmos seqüenciais

Na tabela abaixo, estão resumidos os resultados dos melhores algoritmos seqüenciais que resolvem alguma variante do Problema da Mochila. Eles estão separados em dois grupos, conforme o paradigma em que se baseiam.

| ALGORITMOS | VARIANTE | TEMPO | ESPAÇO |
|---|---|---|---|
| Horowitz e Sahni | KP01 | $\mathcal{O}(2^{n/2})$ | $\mathcal{O}(2^{n/2})$ |
| Schroeppel e Shamir | SSPd | $\mathcal{O}(n2^{n/2})$ | $\mathcal{O}(2^{n/4})$ |
| Bellman | KP | $\mathcal{O}(nc)$ | $\mathcal{O}(nc)$ |
| Yanasse e Soma | SSP | $\mathcal{O}(n(c-2w_{min})+c)$ | $\mathcal{O}(n+c)$ |
| Pisinger | SSP | $\mathcal{O}(nw_{max})$ | $\mathcal{O}(nw_{max})$ |
| Soma e Toth | SSP | $\mathcal{O}((n-\log_2 c^2)(c-2w_{min})+c)$ | $\mathcal{O}(n+c)$ |

Se fôssemos considerar considerar também a recuperação do vetor $X$, seria necessário acrescentar um fator extra $n$ no espaço gasto pelos dois primeiros algoritmos. Pisinger não comentou nada a este respeito com relação ao seu algoritmo.

# Capítulo III

# Algoritmos paralelos para o Problema da Mochila

Vamos apresentar os principais algoritmos paralelos encontrados na literatura científica que resolvem alguma variante do Problema da Mochila na PRAM SIMD. Nessa tarefa de descrição algorítmica, seguiremos uma representação de alto nível semelhante à utilizada no capítulo anterior.

No entanto, em alguns casos utilizaremos um comando que é exclusivo — mas também muito comum — na descrição de algoritmos SIMD: é o **do in parallel**. Considere o exemplo a seguir:

**for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
    *Comando1*
    *Comando2*
    $\vdots$
    *ÚltimoComando*
**end for**

No caso acima, *todos* os processadores $P_i$, $0 \leq i < p$, executam *simultaneamente* e *uma única vez* os comandos presentes no bloco interno do **for**. Além disso, cada processador $P_i$ respeita a ordem desses comandos: executa primeiro o *Comando1*, depois o *Comando2*, e assim por diante.

É possível também selecionar um determinado grupo de processadores para executar cada comando dentro deste bloco. Veja o exemplo abaixo:

**for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
    *Comando1*
    *Comando2*, $(5 < i \leq 8)$

⋮

*ÚltimoComando*

**end for**

Neste caso, o *Comando2* será executado somente pelos processadores $P_5$, $P_6$, $P_7$ e $P_8$. Durante este tempo de execução, os outros processadores não selecionados permanecem ociosos. Os comandos sem seleção continuam sendo executados simultaneamente por todos os processadores.

Feita esta rápida introdução a respeito da notação, passemos agora aos algoritmos paralelos. No esquema a seguir, agrupamos as resoluções de diversas variantes do Problema da Mochila na PRAM SIMD, segundo os dois paradigmas estudados.

- **Paradigma das listas**
  - Karnin (SSPd)
  - Ferreira (SSPd)
  - Chang *et al.* (SSPd)
  - Lou e Chang (SSPd)
  - *Também para o hipercubo:* Ferreira e Robson (SSPd)
- **Paradigma da programação dinâmica**
  - Teng (KP)
  - *Também para o hipercubo:* Lin e Storer (KP01)
  - *Apenas para o hipercubo:*
    - Lee, Shragowitz e Sahni (KP01)
    - Goldman e Trystram (KP)

Como pode ser visto, alguns dos trabalhos fazem referência a implementações numa máquina de topologia de hipercubo. Por este motivo, resolvemos apresentar também os melhores resultados neste modelo, embora não seja alvo específico deste presente estudo.

Apresentaremos agora uma concisa descrição destes algoritmos. Vale a pena frisar que o perfeito entendimento de cada um deles não é essencial para quem apenas deseja compreender os resultados originais desta tese. Neste caso, recomendamos atenção especial a dois trabalhos: Lou e Chang [LoC97] e Lin e Storer [LiS90, LiS91].

## III.1 Algoritmos paralelos segundo o paradigma das listas

Na literatura científica, existem diversos algoritmos que resolvem o Problema da Mochila na PRAM SIMD. Dentre eles, há cinco [Kar84, Fer91, FeR96, Cha94, LoC97] que se baseiam no paradigma das listas, e por isso, como já foi comentado no capítulo anterior, suas complexidades de tempo estão expressas apenas em termos do número $n$ de objetos.

Um dos primeiros algoritmos paralelos para o Problema da Mochila foi o de Karnin [Kar84], que propôs uma paralelização inspirada no *algoritmo das quatro listas* [ScS81]. Seu algoritmo, aplicado a uma PRAM SIMD EREW, encontra a solução do SSPd em tempo $\mathcal{O}(n2^{n/2})$, usando $p = 2.2^{n/6}$ processadores e espaço $\mathcal{O}(2^{n/6})$.

Ferreira [Fer91] apresentou um algoritmo paralelo adaptativo: com $p = \mathcal{O}(2^{n/2})^{(1-\epsilon)}$ processadores, mostra como é possível resolver o SSPd numa PRAM SIMD CREW em tempo $\mathcal{O}(n(2^{n/2})^{\epsilon})$, onde $0 < \epsilon < 1$. Mais tarde, Ferreira e Robson [FeR96] apresentaram um outro algoritmo adaptativo com aceleração ótima em relação ao *algoritmo das quatro listas*: ele gasta tempo $\mathcal{O}(n2^{n/2}/p)$ e espaço $\mathcal{O}(2^{n/4})$ usando $1 \leq p \leq 2^{n/4}$ processadores em uma máquina PRAM SIMD EREW. No entanto, a aceleração deste algoritmo paralelo em relação ao *algoritmo das duas listas* [HoS74] é $\mathcal{O}(p/n)$, ou seja, não é ótima.

Chang *et al.* [Cha94] propuseram uma paralelização para a *fase de geração* do *algoritmo das duas listas* na resolução do SSPd. Segundo afirmavam, esta geração gastaria tempo $\mathcal{O}((n/8)^2)$ numa PRAM SIMD CREW com $p = \mathcal{O}(2^{n/8})$ processadores. No entanto, a análise da complexidade de tempo do algoritmo proposto está incorreta: seu verdadeiro valor é $\mathcal{O}(n2^{n/2})$, como explicaremos adiante.

Lou e Chang [LoC97] deram prosseguimento a este último trabalho. Também utilizando os mesmos $p = \mathcal{O}(2^{n/8})$ processadores, elaboraram uma forma paralela de realizar a *fase de busca* do *algoritmo das duas listas* numa PRAM SIMD CREW. Como resultado final, obtiveram um algoritmo para a resolução do SSPd que gasta tempo $\mathcal{O}(2^{3n/8})$ e que, portanto, teria aceleração $\mathcal{O}(p)$. Entretanto, esse resultado é inválido, pois eles utilizam a *fase de geração* de Chang *et al.*, cuja complexidade de tempo é maior do que supunham.

A partir de agora, vamos apresentar estes resultados de uma forma mais detalhada.

### III.1.1 Algoritmo de Karnin

Karnin [Kar84] apresentou um dos primeiros algoritmos paralelos para o SSPd: inspirando-se nos *heaps* do *algoritmo das quatro listas*, desenvolveu uma paralelização que utiliza $p = 2.2^{n/6}$ processadores. Seu algoritmo também segue as duas conhecidas fases: a geração das listas $A$ e $B$ e a posterior busca. No entanto, o paralelismo ocorre apenas na geração do próximo elemento de cada lista, à medida em que eles vão sendo necessários na *fase de busca*. Esta segunda fase é realizada de modo seqüencial: é simplesmente a mesma busca do *algoritmo das duas listas*. Portanto, o tempo total deste algoritmo será $\mathcal{O}(2^{n/2})$, que é o mesmo tempo seqüencial da *fase de busca*. A diferença está na memória utilizada, como explicaremos no final.

Vamos apresentar apenas o algoritmo paralelo que gera os elementos da lista $A$, pois a geração dos elementos da lista $B$ ocorre de forma análoga.

**Fase de geração:**

1) Divida a primeira metade dos pesos $W_1 = \{w_0, \ldots, w_{n/2-1}\}$ em três partes iguais: $W_{11} = \{w_0, \ldots, w_{n/6-1}\}$, $W_{12} = \{w_{n/6}, \ldots, w_{n/3-1}\}$ e $W_{13} = \{w_{n/3}, \ldots, w_{n/2-1}\}$. Portanto, $|W_{11}| = |W_{12}| = |W_{13}| = n/6$.

2) Crie uma lista $L_1$ que contém, em ordem crescente, todas as possíveis somas dos elementos de $W_{11}$. Chamaremos de $\{a_0, \ldots, a_{2^{n/6}-1}\}$ os elementos ordenados dessa lista $L_1$.

3) Crie uma lista $L_2$ que contém, também em ordem crescente, todas as possíveis somas dos elementos de $W_{12}$. Chamaremos de $\{b_0, \ldots, b_{2^{n/6}-1}\}$ os elementos ordenados dessa lista $L_2$.

4) Crie uma terceira lista $\{c_0, \ldots, c_{2^{n/6}-1}\}$ que contém, em qualquer ordem, todas as possíveis somas dos elementos de $W_{13}$.

5) Crie um *heap* $H$ com os pares $(a_0 + b_0, c_i)$, com $0 \leq i < 2^{n/6}$, que retorna o par cuja soma de seus elementos seja a menor.

Como já foi visto, as três listas acima podem ser criadas em tempo $\mathcal{O}(2^{n/6})$, e o *heap* $H$, formado por $2^{n/6}$ elementos, em tempo $\mathcal{O}(n2^{n/6})$.

Este *heap* $H$ será utilizado para gerar os elementos da lista $A$ à medida em que sejam solicitados pela *fase de busca*. Suponha que no topo de $H$ esteja o par $(t, c_k)$. Dessa forma, o elemento gerado será $t + c_k$, e em seguida se deve inserir em $H$ o par $(\text{prox}(t), c_k)$, onde $\text{prox}(t)$ é o próximo elemento correspondente às listas $L_1$ e $L_2$.

Sabemos que:

$$\text{prox}(t) \quad = \quad \min_{i,j}\{a_i + b_j \mid a_i + b_j > t\}$$

Encontrar este mínimo é equivalente a procurar:

$$\min_{i,j}\{a_i - (t - b_j) \mid a_i - (t - b_j) > 0\}$$

Uma vez que se encontre o valor acima, basta adicioná-lo a $t$ para se obter $\text{prox}(t)$. O algoritmo paralelo apresentado a seguir calcula $\text{prox}(t)$ através da fórmula acima, supondo que as listas $L_1$ e $L_2$ estejam armazenadas um elemento por processador.

**Cálculo de prox$(t)$:**

1) Cada processador, que armazena um elemento $b_j$ da lista $L_2$, calcula $t - b_j$. Isto dará origem a uma nova lista, que chamaremos de $L_3$.

2) Faça um *merge-sort* das listas $L_1$ e $L_3$, criando a lista $PR$, que também estará em ordem crescente. Esta lista terá $2.2^{n/6}$ elementos, um por processador. No caso de que haja valores iguais, o elemento da lista $L_1$ precederá o elemento da lista $L_3$ na ordenação. Chamemos de $PR_i$ o valor da lista $PR$ armazenado pelo $i$-ésimo processador, $0 \leq i < 2.2^{n/6}$. Esse processador também guardará uma outra informação: de qual lista ($L_1$ ou $L_3$) procede seu elemento $PR_i$. Além disso, por convenção, $PR_0 = 0$, e esse elemento será originário da lista $L_3$.

3) Se $PR_i$ procede da lista $L_1$, $PR_{i-1}$ procede da lista $L_3$, e $PR_i - PR_{i-1} > 0$, então faça em paralelo $PR_i \leftarrow PR_i - PR_{i-1}$. Caso contrário, faça $PR_i \leftarrow \infty$.

4) Calcule em paralelo o menor valor de $PR$, e some-o a $t$.

Convém dar algumas explicações sobre o terceiro passo deste algoritmo. Se $PR_i$ contém algum elemento $a_k$ da lista $L_1$, então a subtração de $PR_j$, para $j > i$, gera um resultado negativo que pode ser desprezado. Além disso, para $0 \leq j < i - 1$, o cálculo de $PR_i - PR_j$ também é desnecessário, pois $PR_{i-1}$ é maior do que todos os seus predecessores. Portanto, basta calcular apenas $PR_i - PR_{i-1}$.

Numa PRAM SIMD EREW, o *merge-sort* realizado sobre duas listas de $2^{n/6}$ elementos utilizando-se $p = 2.2^{n/6}$ processadores pode ser feito em tempo $\mathcal{O}(n)$. O terceiro passo gasta tempo constante, pois cada processador comunica-se com apenas um outro processador e realiza somente cálculos aritméticos. O quarto passo pode ser realizado em tempo $\mathcal{O}(n)$.

Como a obtenção de cada elemento das listas $A$ e $B$ gasta tempo $\mathcal{O}(n)$, o tempo total deste algoritmo paralelo é $\mathcal{O}(n2^{n/2})$. No entanto, da mesma forma que [ScS81], Karnin despreza o fator $n$ na complexidade de tempo.

Por outro lado, as listas $L_1, L_2$ e $L_3$ têm tamanho $2^{n/6}$, enquanto a lista $PR$ tem tamanho $2.2^{n/6}$. Como são $2.2^{n/6}$ processadores, cada um deles gasta espaço constante para armazená-las. No entanto, o *heap* $H$ gasta espaço $\mathcal{O}(2^{n/6})$, que é portanto o espaço total necessário para o algoritmo.

Definindo o critério *hardware* $HW$ como $\max(p, M)$, onde $p$ é o número de processadores e $M$ é o número de células de memória utilizadas, no algoritmo de Karnin temos

$HW = \mathcal{O}(2^{n/6})$. Aplicando este mesmo critério aos algoritmos seqüenciais já descritos, teríamos $HW = \mathcal{O}(2^{n/2})$ para o *algoritmo das duas listas* e $HW = \mathcal{O}(2^{n/4})$ para o *algoritmo das quatro listas*. Segundo este parâmetro $HW$, o algoritmo de Karnin representa um resultado significativo.

### III.1.2 Algoritmo de Ferreira

Para a resolução do SSPd, Ferreira [Fer91] propôs uma modificação no *algoritmo das duas listas*, que chamou de *algoritmo seqüencial de uma lista*. Descreveremos sucintamente este novo algoritmo, que tem a vantagem de ser facilmente paralelizável.

**Algoritmo seqüencial de uma lista:**

1) Divida $W$ em duas partes $W_1$ e $W_2$, ambas de tamanho $n/2$.

2) Crie uma lista $A$ que contém, em ordem crescente, todas as possíveis somas de elementos de $W_1$.

3) Enquanto não terminar as possíveis somas de elementos de $W_2$, faça:

   3.1) Encontre a próxima soma $s_2$ de elementos de $W_2$.

   3.2) Faça uma busca binária na lista $A$ para encontrar uma soma $s_1$ tal que $s_1 + s_2 = c$.

   3.3) Se a busca obteve sucesso, então pare: a solução foi encontrada.

4) Pare: não há solução.

Este algoritmo seqüencial gasta tempo $\mathcal{O}(2^{n/2})$ para gerar a lista $A$, e cada busca binária que faz entre os elementos de $A$ gasta tempo $\mathcal{O}(n)$. No pior caso, terá que testar todos as possíveis combinações de elementos de $W_2$, gastando tempo total $\mathcal{O}(n2^{n/2})$. O espaço necessário é $\mathcal{O}(2^{n/2})$.

Ferreira apresentou uma paralelização deste algoritmo para a PRAM SIMD CREW com $p = N^{1-\epsilon}$ processadores, onde $N = 2^{n/2}$ e $0 < \epsilon < 1$, ou seja, quando o número de processadores é menor que o tamanho da lista $A$:

1) Cada processador fica responsável pela geração de $N/N^{1-\epsilon} = N^{\epsilon}$ somas da lista $A$, gastando para isso tempo $\mathcal{O}(N^{\epsilon} \log N)$.

2) É feita uma ordenação desses $N$ elementos utilizando-se $N^{1-\epsilon}$ processadores em tempo $\mathcal{O}(N^{\epsilon} \log N)$ (*cfr.* [Akl84]).

3) Na última fase, cada processador deve fazer, no pior caso, $N/N^{1-\epsilon} = N^{\epsilon}$ buscas binárias, gastando tempo total de $\mathcal{O}(N^{\epsilon} \log N)$.

Portanto, o tempo total deste algoritmo é $\mathcal{O}(N^{\epsilon} \log N) = \mathcal{O}(n2^{n\epsilon/2})$, onde $0 < \epsilon < 1$. São utilizados $p = \mathcal{O}(2^{n/2})^{(1-\epsilon)}$ processadores e o espaço gasto é $\mathcal{O}(2^{n/2})$. Este é o primeiro algoritmo paralelo adaptativo que resolve uma variante do Problema da Mochila numa PRAM SIMD através do paradigma das listas.

### III.1.3 Algoritmo de Ferreira e Robson

O principal resultado do trabalho de Ferreira e Robson [FeR96] é uma paralelização adaptativa do *algoritmo das quatro listas* [ScS81], elaborada principalmente para a arquitetura do hipercubo, embora possa ser utilizada em outros modelos. *Quando for possível realizar uma ordenação paralela em tempo logarítmico*, este algoritmo alcança aceleração ótima com relação ao *algoritmo das quatro listas*: resolve o SSPd em tempo $\mathcal{O}(n2^{n/2}/p)$ e espaço $\mathcal{O}(2^{n/4})$, usando $1 \leq p \leq 2^{n/4}$ processadores[1]. É importante ressaltar que, mesmo nesse caso, sua aceleração com relação ao *algoritmo das duas listas* não é ótima.

Este trabalho baseia-se principalmente em estruturas chamadas multiconjuntos ordenados (*sorted multisets*). Especificamente, os autores trabalham com somas em vetores ordenados: se $X$ e $Y$ são vetores ordenados de dimensão $m$, dizemos que $z$ está em $X+Y$ se existirem índices $i$ e $j$ tais que $z = X[i] + Y[j]$, onde $1 \leq i, j \leq m$. Sendo $X$,$Y$,$R$ e $S$ vetores ordenados de tamanho $O(m)$, Ferreira e Robson apresentaram dois algoritmos paralelos que geram os elementos de $X+Y+R+S$ gastando espaço $O(m)$. Torna-se fácil obter um algoritmo paralelo para a resolução do SSPd quando consideramos esses quatro vetores como as correspondentes listas do algoritmo de Schroeppel e Shamir [ScS81], onde $m = 2^{n/4}$.

O primeiro algoritmo paralelo de Ferreira e Robson baseia-se em *heaps*: gasta tempo $\mathcal{O}(n2^{n/2}/p)$ usando $1 \leq p \leq 2^{n/4}$ processadores, mas exige espaço $\mathcal{O}(2^{n/4})$ por processador. Não iremos comentá-lo, pois o segundo algoritmo, que trabalha com amostragens ao invés dos *heaps*, é mais eficiente: gasta tempo $\mathcal{O}(\sigma(2^{n/2})n2^{n/2}/p)$ e espaço total $\mathcal{O}(2^{n/4})$, usando $1 \leq p \leq 2^{n/4}$ processadores. $\sigma(m)$ é chamado de *sorting overhead*, ou seja, é o fator de tempo que excede $O(\log m)$ na ordenação paralela de $O(m)$ elementos, quando cada processador armazena inicialmente $O(1)$ elementos. Por exemplo, se o algoritmo de ordenação de Cole [Col88] for utilizado, temos $\sigma(m) = 1$ para qualquer $m$, e portanto

[1]Seguiremos também na descrição deste algoritmo a sua notação original, ou seja, os índices dos processadores, objetos, pesos e lucros começam em 1 ao invés de 0.

o algoritmo de Ferreira e Robson pode resolver o SSPd numa PRAM SIMD EREW em tempo $O(n2^{n/2}/p)$. No entanto, quando se utilizam algoritmos paralelos de ordenação tradicionalmente mais práticos — como o de Batcher [Bat68], por exemplo, no caso de uma implementação no hipercubo —, o tempo passa a ser $O(n^2 2^{n/2}/p)$.

Vamos apresentar resumidamente o algoritmo paralelo por amostragens. Sendo $X$ e $Y$ vetores ordenados de tamanho $m$, a idéia principal é gerar $X+Y$ como uma seqüência de segmentos, todos eles de tamanho $\mathcal{O}(m)$. Os limites desses segmentos são obtidos através da geração recursiva de uma amostragem de $X+Y$, selecionando dela um subconjunto (chamado de *esqueleto*) tal que o número de elementos de $X+Y$ que estejam entre dois elementos sucessivos desse esqueleto seja garantidamente $\mathcal{O}(m)$. Por simplicidade, supõe-se que $m$ seja uma potência de 2.

Inicialmente, descreveremos um algoritmo seqüencial chamado *GeradordeSegmento* que encontra os elementos de $X+Y$ presentes entre dois limites pré-estabelecidos. Ele funciona da seguinte forma: identifica todos os pares $i, j$ tais que $X[i]+Y[j]$ esteja entre esses limites, armazena tais pares de acordo com sua ordem lexicográfica, calcula as respectivas somas $X[i]+Y[j]$ e, por fim, ordena-as.

A estrutura básica deste procedimento está indicada a seguir. Ele gera o segmento ordenado formado pelos elementos de $X+Y$ que estejam no intervalo $(min, max]$, retornando na variável *total* o número desses elementos, que ficarão armazenados nas primeiras posições do vetor *Res*.

*GeradordeSegmento* $(min, max, Res[], total)$
    *Identifica* $(minj[], maxj[])$
    *Localiza* $(minloc[], maxloc[], total)$
    *Distribui* $(minloc[], maxloc[], Xval[], jval[])$
    *Calcula* $(Xval[], jval[], Res[])$
    *Ordena* $(Res[])$

O procedimento *Identifica* constrói os vetores $minj$ e $maxj$. Para cada índice $i$ de $X$ que esteja no intervalo considerado, dois registros são construídos, contendo respectivamente $(X[i], i)$ e $(min - Y[i], 0)$. Após estas duas seqüências monotônicas serem intercaladas (considerando os valores da sua primeira componente), se um elemento $(X[i], i)$ estiver na posição *pos* da seqüência final, então $X[i]$ será maior do que exatamente $pos - i$ valores $min - Y[j]$. Portanto, o valor de $minj[i]$ deverá ser $m - (pos - i)$. A construção do vetor $maxj$ a partir do valor $max$ é análoga.

*Localiza* calcula, para cada $i$, $minloc[i]$ e $maxloc[i]$, que são os $ranks$ de $(i, minj[i])$ e $(i, maxj[i])$ na ordem lexicográfica obtida. Este procedimento calculará também o número total de somas dentro do intervalo.

*Distribui* calcula os valores de $X[i]$ e $j$ para cada soma $X[i]+Y[j]$ dentro do intervalo, que serão armazenados em dois vetores chamados $Xval$ e $jval$, de acordo com a ordem lexicográfica de seus pares $(i, j)$.

*Calcula* fará as somas $X[i] + Y[j]$, armazenando-as no vetor $Res$ seguindo a mesma ordem dos seus valores $X[i], j$. Por fim, *Ordena* deixará esses valores em ordem crescente.

Todos esses procedimentos seqüenciais, com exceção da ordenação final, podem ser executados em tempo e espaço $\mathcal{O}(m)$, desde que o número de elementos no segmento seja também $\mathcal{O}(m)$. A ordenação, como é sabido, exige tempo $\mathcal{O}(m \log m)$, que é portanto o tempo total gasto pelo *GeradordeSegmento*.

Para garantir que os segmentos gerados pelas chamadas ao algoritmo *GeradordeSegmento* tenham de fato $\mathcal{O}(m)$ elementos, os autores utilizam um resultado conhecido como *Sample Rank Lemma* [MiA85]. Dados dois vetores ordenados $X$ e $Y$ de tamanho $m$, chama-se *rank* de uma soma qualquer $X[i] + Y[j]$, $1 \leq i, j \leq m$, a posição que essa soma ocuparia se todos os elementos de $X+Y$ fossem ordenados (os pares com somas iguais são ordenados lexicograficamente de acordo com os valores de seus índices $i$ e $j$). De modo análogo, o *sample rank* de um elemento é o *rank* que ele tem na amostragem. Portanto, considerando uma amostragem com separação $d$, onde $d$ é um fator de $m$, o *sample rank* $s$ e o *rank* $r$ de um mesmo elemento dessa amostragem satisfazem a seguinte inequação:

$$sd^2 \leq r \leq sd^2 + (2m/d - 2)(d^2 - 1).$$

A partir deste lema, é possível observar o seguinte fato: considerando dois elementos sucessivos de uma mesma amostragem — isto é, com *sample ranks* $s$ e $s+1$ respectivamente —, o número de elementos presentes entre eles será no máximo $(s+1)d^2+(2m/d-2)(d^2-1) - sd^2 = d^2 + (2m/d - 2)(d^2 - 1) < d^2 + (2m/d)d^2 = d^2 + 2md$.

Apoiando-se neste resultado, os autores apresentaram um algoritmo seqüencial recursivo chamado *Gerador*, que recebe como parâmetros os vetores $X$ e $Y$, seu tamanho $m$, a variável *pact* e um procedimento *Ação*. *pact* é um fator de $m$ que define a *parte ativa* dos vetores; concretamente, os *pact* elementos cujos índices são múltiplos de $m/pact$. Na primeira chamada do procedimento *Gerador*, $pact = m$. Como resultado deste algoritmo, são gerados todos os elementos de $X + Y$ cujas parcelas estão nas correspondentes partes ativas de $X$ e $Y$. Esses elementos são gerados em uma seqüência de um ou mais segmentos

ordenados, cada um deles de tamanho $\mathcal{O}(m)$, e cada elemento em qualquer segmento é menor do que qualquer elemento de um segmento posterior. Assim que cada segmento é gerado, o procedimento *Ação* é aplicado nele.

A descrição do procedimento *Gerador* está abaixo:

*Gerador* $(X, Y, m, pact$,*Ação*$)$
**if** $pact \leq m$ **then**
    $GeraçãoSimples(m, pact, Res[])$
    $Ação(Res[], pact^2)$
**else**
    **if** $pact = m$ **then** $pact' \leftarrow m/2$
        **else** $pact' \leftarrow pact^2/m$
    **endif**
    $Gerador\ (X, Y, m, pact', EscolhaEsqueleto)$
    $esqueleto' \leftarrow esqueleto$
    $contador \leftarrow 0$
    **for** $i \leftarrow 1$ **to** $pact'$ **do**
        $GeradordeSegmento\ (esqueleto'[i-1], esqueleto'[i], Res[], total)$
        $Ação\ (Res[], total)$
    **end for**
**end if**

Como a amostragem é formada por $pact$ elementos de cada vetor, ela contém $pact^2$ somas. Se esse número não ultrapassa o valor de $m$, as somas são calculadas diretamente pelo algoritmo *GeraçãoSimples* como um único segmento, no qual o procedimento *Ação* é posteriormente aplicado. Caso contrário, é escolhido um conjunto menor formado por $pact'$ elementos tanto de $X$ como de $Y$, e o procedimento *Geração* é chamado recursivamente sobre ele, gerando uma amostragem de tamanho $pact'^2$. A variável *contador* contabiliza o número de somas da amostragem geradas até então. Esta variável é utilizada e atualizada pelo procedimento *EscolhaEsqueleto*, que é passado como parâmetro. Sua função é simplesmente armazenar cada elemento da amostragem na variável $esqueleto[]$, que terá tamanho $pact'$. Esta variável é copiada em outro vetor $esqueleto'$ para evitar sua corrupção durante as chamadas dos procedimentos *Ação* (que serão sempre *Escolha-Esqueleto*, com exceção do primeiro nível).

Por fim, o conjunto $X + Y$ é gerado a partir dos $pact'$ segmentos que ficam entre

elementos adjacentes da variável *esqueleto*′. Nestas chamadas de *GeradordeSegmento*, somente são utilizadas as partes ativas de $X$ e $Y$, e considera-se que $esqueleto[0] = -\infty$.

Passemos agora para a análise deste algoritmo seqüencial. É possível observar que a amostragem tem separação $d = pact/pact'$, e o número de elementos em cada vetor é $pact$. Portanto, considerando o resultado do *Sample Rank Lemma* apresentado logo acima, sabemos que o tamanho dos segmentos será no máximo $(pact/pact')^2 + 2pact^2/pact'$, que é menor que $3pact^2/pact'$, pois $pact' > 1$. Devido ao modo como $pact'$ é escolhido no algoritmo, é fácil verificar que $pact^2/pact' \leq 2m$. Portanto, os segmentos terão tamanho menor que $6m$, ou seja, serão $\mathcal{O}(m)$. Uma vez que cada esqueleto é formado por exatamente $pact'$ elementos, o espaço utilizado também será $\mathcal{O}(m)$.

Excluindo a chamada recursiva, o tempo gasto pelo algoritmo *Gerador* corresponderá às $pact'$ chamadas de *GeradordeSegmento*, num total de $\mathcal{O}(pact'm \log m)$. Considerando todas as chamadas recursivas a partir de $pact = m$, o tempo será $\mathcal{O}(m \log m \sum pact')$. Como os valores de $pact'$ são potências de 2 distintas e menores do que $m$, temos que $\sum pact' < 2m$. Portanto, o tempo total deste algoritmo será $\mathcal{O}(m^2 \log m)$.

Embora os autores tenham elaborado este algoritmo seqüencial para ser paralelizado numa arquitetura de hipercubo, ele também é válido para outros modelos. Supõe-se que inicialmente os vetores $X$ e $Y$ estejam armazenados um elemento por processador. Uma convenção é adotada para o caso em que se deseja armazenar um conjunto de $m$ elementos em $p$ processadores, quando $p \neq m$: considerando a densidade $\delta$ deste conjunto como a primeira potência $2^i$ maior ou igual a $m/p$, para $i$ natural, o conjunto estará armazenado com $\delta$ elementos em cada um dos primeiros $\lceil m/\delta \rceil$ processadores, $m$ mod $\delta$ no próximo, e nenhum nos restantes. O conjunto é considerado ordenado se e somente se os elementos em cada processador estiverem ordenados e cada valor presente no processador $i$ for menor ou igual a qualquer valor no processador $j$, para $i < j$.

Na paralelização do algoritmo *GeradordeSegmento*, os autores utilizaram operações desenvolvidas especificamente para o hipercubo (*Concentrate, Generalize, Monotonic Route, Prefix Sum, Bitonic Merge, Sort* e *Random Read*), cujas descrições e análises não apresentam no artigo (remetem para [Fer96]). Essas operações, aplicadas a conjuntos de tamanho $\mathcal{O}(m)$, gastam tempo $\mathcal{O}(\sigma(m) \log m)$. Com elas, é possível obter uma versão paralela de *GeradordeSegmento* que gasta tempo $\mathcal{O}(\sigma(m) \log m)$ desde que $\delta = \mathcal{O}(1)$, que é o caso considerado. Além disso, o espaço usado em cada processador é $\mathcal{O}(1 + \delta) = \mathcal{O}(1)$.

Com isso, o algoritmo *Gerador* torna-se praticamente um algoritmo paralelo: os subvetores de tamanho *pact* na versão seqüencial correspondem a conjuntos de processadores

ativos, que por sua vez são sub-hipercubos. Como os vetores $X$ e $Y$ têm tamanho $\mathcal{O}(m)$ e estão distribuídos sobre $m$ processadores, e como qualquer esqueleto contém exatamente $pact'$ elementos, então no máximo um elemento do esqueleto será armazenado em cada processador. Portanto, o espaço total utilizado em cada processador será $\mathcal{O}(1)$.

Podemos agora calcular o tempo de execução de *Gerador*. Se excluirmos as suas recursões, o tempo gasto corresponderá a $pact'$ chamadas a *GeradordeSegmento*, ou seja, será $\mathcal{O}(pact'\sigma(m)\log m)$. Portanto, o tempo total, onde incluímos todas as suas chamadas recursivas, será $\mathcal{O}(\sigma(m)\log m \sum pact') = (m\sigma(m)\log m)$.

Este algoritmo é adaptativo: se houver menos do que $m$ processadores disponíveis, cada processador poderá simular $m/p$ processadores virtuais, gastando assim tempo total $\mathcal{O}((m^2\sigma(m)\log m)/p)$ e espaço total $\mathcal{O}(m)$.

Voltando à resolução do SSPd através do uso de quatro listas, que correspondem a quatro vetores ordenados de tamanho $2^{n/4}$, a utilização deste algoritmo *Gerador* permite encontrar a solução em tempo $\mathcal{O}(n2^{n/2}\sigma(2^{n/4})/p)$ numa máquina paralela com $1 \leq p \leq 2^{n/4}$ processadores, exigindo espaço total $\mathcal{O}(2^{n/4})$. Se essa máquina for um hipercubo e utilizarmos um algoritmo prático de ordenação paralela, temos $\sigma(2^{n/4}) = n/4$, e portanto o tempo de resolução será $\mathcal{O}(n^2 2^{n/2}/p)$.

Por outro lado, se utilizarmos o algoritmo de ordenação de Cole [Col88] numa PRAM SIMD EREW, esse tempo será $\mathcal{O}(n2^{n/2}/p)$. Curiosamente, os autores parecem não conhecer este algoritmo paralelo de ordenação, pois afirmam que tal tempo somente poderia ser obtido numa PRAM SIMD CRCW.

### III.1.4 Algoritmo de Chang *et al.*

Como já dissemos anteriormente, o algoritmo de Chang *et al.* [Cha94] procura resolver o SSPd gerando as duas listas $A$ e $B$ de forma paralela numa PRAM SIMD CREW com $p = \mathcal{O}(2^{n/8})$ processadores. O algoritmo é composto por quatro passos, e é semelhante para cada lista; por isso, somente descreveremos a criação da lista $A$.

**Passo 1:**

A primeira metade dos pesos $W_1 = \{w_0, w_1, \ldots, w_{n/2-1}\}$ é dividida em três partes: $W_{11} = \{w_0, w_1, \ldots, w_{n/8-1}\}$ e $W_{12} = \{w_{n/8}, w_{n/8+1}, \ldots, w_{n/4-1}\}$ de mesmo tamanho, e $W_{13} = \{w_{n/4}, w_{n/4+1}, \ldots, w_{n/2-1}\}$, cujo tamanho é o dobro das anteriores. Portanto, $|W_{11}| = |W_{12}| = |W_{13}|/2 = n/8$.

**Passo 2:**

Em paralelo, todas as somas entre os elementos de $W_{11}$, $W_{12}$ e $W_{13}$ são calculadas, ordenadas e armazenadas respectivamente nos vetores $X$, $Y$ e $Z$. Dessa forma, $|X| = |Y| = |Z|^{1/2} = 2^{n/8}$.

**Passo 3:**

1) Associe $X[i]$, $Y$ e $Z[0]$ ao processador $P_i$, $0 \leq i < p$.

2) Cada processador $P_i$, $0 \leq i < p$, calcula os vetores $R_i$ e $S_i$ tais que $R_i[j] = X[i] + Y[j]$ e $S_i[j] = R_i[j] + Z[0]$, para $0 \leq j < 2^{n/8}$.

3) Cada processador $P_i$, $0 \leq i < p$, contém um vetor $INDZ_i$, cuja posição $j$ indica qual o elemento de $Z$ que está presente na somatória definida em $S_i[j]$. No início, $INDZ_i[j] = 0$, para todo $j$.

4) Cada processador $P_i$, $0 \leq i < p$, mantém atualizado $MINS_i$, que contém o mínimo valor do vetor $S_i$, e também $v_i$, que é a posição de $MINS_i$ em $S_i$.

**Passo 4:**

$a \leftarrow 0$
**while** $a < 2^{n/2}$ **do**
    $a \leftarrow a + 1$
    $A[a] \leftarrow min\{MINS_i, 0 \leq i < 2^{n/8}\}$
    Seja $u$ o índice do processador que contém $\min\{MINS_i\}$
    $INDZ_u[v_u] \leftarrow INDZ_u[v_u] + 1$
    **if** $INDZ_u[v_u] > 2^{n/4} - 1$ **then** $S_u[v_u] \leftarrow \infty$
                **else** $S_u[v_u] \leftarrow R_u[v_u] + Z[INDZ_u[v_u]]$
    **end if**
**end while**

Convém esclarecer que este passo está incompleto no artigo original: falta o teste feito pelo comando **if**, sem o qual o algoritmo não funciona.

**Análise da complexidade de tempo**

O **Passo 1** pode ser feito em tempo $\mathcal{O}(n/4)$. O **Passo 2**, utilizando um algoritmo descrito em [Akl89], gasta tempo $\mathcal{O}((n/8)^2)$ com $p = \mathcal{O}(2^{n/8})$ processadores. No **Passo 3** e no

**Passo 4**, o tempo gasto é dominado pelas buscas dos mínimos valores. Chang *et al.* afirmam que ambos os passos podem ser realizados em tempo $\mathcal{O}(\log 2^{n/8})$, o que é um absurdo, pois no **Passo 4** há um laço com $2^{n/2}$ iterações.

Cada processador $P_i$, $0 \leq i < p$, armazena um vetor $S_i$ com $2^{n/8}$ posições, cujos elementos vão sendo trocados à medida que o algoritmo se desenvolve. Além disso, mantém a informação de qual é o menor elemento em cada iteração e sua posição em $S_i$. Parece-nos ser conveniente que em cada processador $P_i$ haja uma estrutura de *heap* para armazenar esses elementos: assim a tarefa de inserção/remoção gastará tempo $\mathcal{O}(\log 2^{n/8})$, mantendo atualizada a informação do valor mínimo de cada $S_i$. Além disso, encontrar o menor entre os $MINS_i$, onde $0 \leq i < p$ e $p = \mathcal{O}(2^{n/8})$, também gasta tempo $\mathcal{O}(\log 2^{n/8})$. Portanto, o tempo total deste algoritmo é $\mathcal{O}((n/8)^2) + \mathcal{O}(2^{n/2} \log 2^{n/8}) = \mathcal{O}(n2^{n/2})$.

Dessa forma, este algoritmo resolve o SSPd em tempo $\mathcal{O}(2^{n/2}) + \mathcal{O}(n2^{n/2}) = \mathcal{O}(n2^{n/2})$, utilizando $p = \mathcal{O}(2^{n/8})$ processadores. Em outras palavras: em termos de complexidade de tempo, ele é pior que o *algoritmo das duas listas*. É fácil observar porque isto ocorre: os elementos de $A$ e $B$ são gerados de uma maneira inerentemente seqüencial.

Além disso, é importante frisar que os autores também afirmam que este algoritmo gasta espaço $\mathcal{O}(2^{n/4})$. Se isso fosse verdade, então nem a lista $A$ e nem a lista $B$ poderiam ser armazenadas na memória, o que inviabilizaria a subseqüente *fase de busca* do *algoritmo das duas listas*, que eles utilizam logo em seguida. Na realidade, o espaço gasto também deve ser $\mathcal{O}(2^{n/2})$, ou seja, o mesmo do algoritmo seqüencial.

A constatação destes erros, além de inviabilizar os resultados do algoritmo que será apresentado a seguir, também possibilitou-nos a publicação de um artigo [SSY02].

### III.1.5 Algoritmo de Lou e Chang

Lou e Chang [LoC97] perceberam que o algoritmo de Chang *et al.*, na resolução do SSPd, não tira nenhum proveito dos processadores na subseqüente *fase de busca*. Por isso, conceberam um algoritmo paralelo que realiza esta última fase em tempo $\mathcal{O}(2^{3n/8})$ na mesma máquina PRAM SIMD CREW com $p = \mathcal{O}(2^{n/8})$. No entanto, as complexidades finais apresentadas neste trabalho também não são válidas, uma vez que os autores utilizam a *fase de geração* de Chang *et al.* que, como foi visto acima, gasta tempo $\mathcal{O}(n2^{n/2})$. De qualquer forma, a paralelização da *fase de busca* é correta e bastante interessante.

Vamos supor que as duas listas $A$ e $B$ já foram geradas. Por isso, podemos dividi-las, cada uma delas, em $p = 2^{n/8}$ blocos de tamanho $2^{n/2}/2^{n/8} = 2^{3n/8} = e$, de tal forma que os menores elementos de $A$ e os maiores elementos de $B$ fiquem nos blocos de menor índice.

Dessa forma, temos a lista $A$ formada por $p$ blocos $A_i$, $0 \leq i < p$, onde $A_i[j] = A[ie+j]$, $0 \leq j < e$. Podemos vê-la como $A = [A_0 \vdots A_1 \vdots \ldots \vdots A_{p-1}]$ e, por hipótese, $A_i[j] \leq A_i[k]$, $0 \leq i < p$, $0 \leq j \leq k < e$. A mesma convenção é utilizada para a lista $B$, com a única diferença que $B_i[j] \geq B_i[k]$, $0 \leq i < p$, $0 \leq j \leq k < e$.

A solução do SSPd poderá estar em um par de blocos $(A_i, B_j)$ se houver pelo menos um elemento $A_i[k]$ e outro $B_j[l]$ tais que $A_i[k] + B_j[l] = c$, $0 \leq i, j < p$, $0 \leq k, l < e$. Com estas observações, é possível tirar duas conclusões simples e importantes:

**Lema 1:** Se $A_i[0] + B_j[e-1] > c$, então a solução do problema não pode estar no par $(A_i, B_j), 0 \leq i, j < p$. Em outras palavras: se a soma dos menores elementos de $A_i$ e de $B_j$ forem maiores do que $c$, então qualquer somatória entre elementos de $A_i$ e $B_j$ também será maior.

**Lema 2:** Se $A_i[e-1] + B_j[0] < c$, então a solução do problema não estará no par $(A_i, B_j), 0 \leq i, j < p$. É uma situação análoga à anterior, mas para os maiores elementos de $A_i$ e $B_j$.

A partir destes dois lemas, os autores conceberam uma fase intermediária entre a geração das listas e a busca: é a chamada *fase de descarte*. Nesta fase, os processadores trabalham simultaneamente procurando descobrir quais os pares de blocos que podem conter a solução do problema.

Cada bloco $A_i$ é associado ao processador $P_i$, $0 \leq i < p$, que recebe a incumbência de buscar em toda a lista $B$ os blocos que, junto com $A_i$, podem conter a solução do problema. Através da aplicação dos dois lemas acima, $P_i$ pode selecionar pares de blocos $(A_i, B_j)$, $0 \leq j < p$, e escrevê-los em uma determinada área da memória compartilhada. Como estamos trabalhando no modelo CREW, todos os processadores podem ler simultaneamente as posições da lista $B$. Portanto, a complexidade de tempo desta fase é $\mathcal{O}(2^{n/8})$, pois há $p = 2^{n/8}$ processadores, e cada um deles testa $p$ blocos $B_j$'s. Abaixo está a descrição do algoritmo concebido para a *fase de descarte*:

**for** $i \leftarrow 0$ **to** $2^{n/8} - 1$ **do in parallel**
  **for** $j \leftarrow 0$ **to** $2^{n/8} - 1$ **do**
    $x \leftarrow A_i[0] + B_j[e-1]$
    $y \leftarrow A_i[e-1] + B_j[0]$
    **if** $(x = c)$ **or** $(y = c)$ **then** solução encontrada
      **else if** $(x < c)$ **and** $(y > c)$ **then**
        escreva $(A_i, B_j)$ na memória compartilhada
      **end if**

**end if**
**end for**
**end for**

Em seguida, Lou e Chang mostraram que o número de pares selecionados por esta *fase de descarte* não pode ser muito grande. Na verdade, os próximos dois lemas garantem que o número de pares $(A_i, B_j), 0 \leq i, j < p$, é no máximo $2.2^{n/8}$.

**Lema 3:** Se os pares de blocos $(A_i, B_{j_1}), (A_i, B_{j_2}), \ldots, (A_i, B_{j_m})$ foram selecionados pelo processador $P_i$, $0 \leq i < p$; $0 \leq j_k < p$; $1 \leq k \leq m$; então $B_{j_1}, B_{j_2}, \ldots, B_{j_m}$ são adjacentes um a um.

**Demonstração:** Suponha que o processador $P_i$ selecione o bloco $B_j$, mas não os blocos $B_{j-1}$ e $B_{j+1}$. Logo, $A_i[0] + B_{j-1}[e-1] > c$ e $A_i[e-1] + B_{j+1}[0] < c$. Para $k < j-1$, sabemos que $B_k[e-1] > B_{j-1}[e-1]$. Logo, $A_i[0] + B_k[e-1] > c$, ou seja, $B_k$ não foi selecionado por $P_i$. De maneira análoga, $B_k[0] < B_{j+1}[0]$, para $k > j+1$. Logo, $A_i[e-1] + B_k[0] < c$, ou seja, $B_k$ não foi selecionado por $P_i$. Conclusão: se $B_{j-1}$ e $B_{j+1}$ não foram selecionados por $P_i$, então mais nenhum outro bloco o foi. □

**Lema 4:** A *fase de descarte* seleciona, no máximo, $2.2^{n/8}$ pares de blocos $(A_i, B_j), 0 \leq i, j < p$.

**Demonstração:** Suponha que o processador $P_i$ selecione os pares de blocos $(A_i, B_j)$, $(A_i, B_{j+1}), \ldots, (A_i, B_{k-1}), (A_i, B_k)$. Então, $A_i[e-1] + B_k[0] > c$. Além disso, devido às ordenações das listas $A$ e $B$, sabemos que $A_i[e-1] \leq A_{i+1}[0]$, e que $B_k[0] \leq B_t[e-1]$, para $j \leq t < k$. Portanto, $A_{i+1}[0] + B_t[e-1] > c$, para $j \leq t < k$, ou seja, $P_{i+1}$ descartará os pares $(A_{i+1}, B_t)$, para $j \leq t < k$. Conclusão: somente um único bloco $B_k$ pode ter sido selecionado simultaneamente por $P_i$ e $P_{i+1}$. Portanto, no máximo $2.2^{n/8}$ pares são selecionados no total. □

Uma vez terminada a *fase de descarte*, a *fase de busca* resume-se em dividir entre os processadores os pares de blocos selecionados, para que cada um em paralelo procure a solução final. Como foram gerados no máximo $2.2^{n/8}$ pares de blocos, cada processador deverá executar no máximo duas vezes a *fase de busca* seqüencial em pares de tamanho $e = 2^{3n/8}$. Dessa forma, a *fase de busca* paralela gasta tempo $\mathcal{O}(2^{3n/8})$.

É importante ressaltar que, embora a *fase de descarte* esteja correta em [LoC97], ela está **incompleta**, pois os autores não mencionam como os blocos selecionados são escritos na memória compartilhada e como é feita a posterior alocação destes pares de blocos pelos processadores.

**Análise da complexidade de tempo**

*Fase de geração:* Lou e Chang afirmam que seria $\mathcal{O}((n/8)^2)$, pois utilizam o algoritmo de Chang *et al.* [Cha94]. No entanto, como vimos anteriormente, a complexidade deste algoritmo é $\mathcal{O}(n2^{n/2})$.

*Fase de descarte:* Gasta tempo $\mathcal{O}(2^{n/8})$.

*Fase de busca:* Gasta tempo $\mathcal{O}(2^{3n/8})$.

Como Lou e Chang não perceberam o erro de análise de Chang *et al.*, calcularam a complexidade do seu algoritmo como $\mathcal{O}(2^{3n/8})$, o que lhe daria uma aceleração $\mathcal{O}(p)$, uma vez que $p = \mathcal{O}(2^{n/8})$. Na verdade, a complexidade final de tempo do algoritmo de Lou e Chang é $\mathcal{O}(n2^{n/2})$, e sua aceleração passa a ser $\mathcal{O}(1/\log p)$.

Ainda apoiando-se nos resultados de Chang *et al.*, os autores também afirmam que seu algoritmo gasta apenas espaço $\mathcal{O}(2^{n/4})$. No entanto, a *fase de descarte* exige que ambas as listas $A$ e $B$ estejam previamente geradas e divididas entre os $p = \mathcal{O}(2^{n/8})$ processadores, ou seja, seu espaço total passa a ser também $\mathcal{O}(2^{n/2})$, que é o tamanho das listas.

Dessa forma, ambas as análises de complexidade de tempo e de espaço publicadas por Lou e Chang [LoC97] estão **incorretas**, conforme descrito no nosso artigo já publicado [SSY02].

## III.2 Algoritmos paralelos segundo o paradigma da programação dinâmica

A partir de agora, comentaremos os trabalhos que apresentam algoritmos paralelos para a resolução do Problema da Mochila na PRAM SIMD a partir do paradigma da programação dinâmica. Na verdade, encontramos apenas dois artigos: Teng [Ten90] e Lin e Storer [LiS91]. O primeiro apresenta três algoritmos paralelos que resolvem o KP em casos particulares do número de processadores, enquanto o outro artigo, cujo resultado também foi apresentado numa conferência internacional [LiS90], contém o único algoritmo adaptativo de toda a literatura que resolve o KP01 numa PRAM SIMD através do paradigma da programação dinâmica. Por isso, este será o resultado que iremos apresentar com mais detalhe.

Durante nossa pesquisa bibliográfica, também encontramos outros trabalhos baseados neste paradigma, mas dirigidos para a topologia de hipercubo: o algoritmo de Lee, Shragowitz e Sahni [LSS88] para o KP01, e dois algoritmos de Goldman e Trystram

[GoT97, GoT03] para o KP. Manteremos um procedimento semelhante ao realizado no paradigma anterior, comentando-os brevemente.

A apresentação destes trabalhos seguirá a ordem cronológica de publicação. No entanto, voltamos a frisar que, dentre todos eles, apenas o trabalho de Lin e Storer [LiS90, LiS91] tem relação direta com os resultados desta tese, pois é o único a apresentar um algoritmo paralelo adaptativo para a resolução de alguma variante do Problema da Mochila na PRAM SIMD.

### III.2.1 Algoritmo de Lee, Shragowitz e Sahni

Lee, Shragowitz e Sahni [LSS88] foram os primeiros a apresentar um algoritmo paralelo adaptativo para a resolução do KP01 através do paradigma da programação dinâmica. É um trabalho voltado para o hipercubo de $p$ processadores e, considerando as definições da subseção II.3.1, resume-se em três fases:

(1) *Decomposição.* KNAP$(V, c)$ é particionado em $p$ subproblemas KNAP$(V_i, c)$, $0 \leq i < p$, tais que $V = \bigcup_{i=0}^{p-1} V_i$, $V_i \cap V_j = \emptyset$ se $i \neq j$, e $|V_i| = |V|/p$ para todo $i$.

(2) *Programação dinâmica.* Cada um desses subproblemas KNAP$(V_i, c)$ é associado a um único processador $P_i$, $0 \leq i < p$, que o resolverá através do algoritmo seqüencial de Bellman.

(3) *Combinação.* As soluções ótimas para os subproblemas encontradas na fase anterior são combinadas entre si para se obter a solução final de KNAP$(V, c)$.

Os autores supõem que a fase de decomposição seja realizada no momento em que o problema é carregado no hipercubo, não exigindo dessa forma nenhum tempo de processamento. Como é fácil perceber que a segunda fase gasta tempo $\mathcal{O}(nc/p)$, a maior dificuldade deste algoritmo está na combinação das soluções, que, na sua implementação, utiliza a observação descrita a seguir.

Considerando os problemas KNAP$(R, c)$ e KNAP$(S, c)$, onde $R \cap S = \emptyset$, sejam $F_R(c) = (f_R(0), f_R(1), \ldots, f_R(c))$ e $F_S(c) = (f_S(0), f_S(1), \ldots, f_S(c))$ os seus respectivos vetores de lucro ótimo. Portanto, o vetor de lucro ótimo do problema KNAP$(T = R \cup S, c)$ será $F_T(c) = (f_T(0), f_T(1), \ldots, f_T(c))$ onde:

$$f_T(i) = \max_{0 \leq j \leq i}\{f_R(j) + f_S(i-j)\}, \text{ para } 0 \leq i \leq c.$$

Essa combinação de soluções aplica-se a apenas dois subproblemas e requer tempo seqüencial $\mathcal{O}(c^2)$. Para encontrar a solução ótima final correspondente a todos os subproblemas, os autores utilizaram uma árvore binária de combinações.

Não detalharemos a implementação desta fase de combinação, embora não tenha em si grandes dificuldades; de qualquer forma, sua complexidade de tempo é $\mathcal{O}(c \log p + c^2)$, onde $p \leq n$. Portanto, o tempo total exigido pelo algoritmo paralelo de Lee *et al.* é $\mathcal{O}(\frac{nc}{p} + c \log p + c^2)$. Os autores também cuidam de manter as informações necessárias para recuperar o vetor binário $X$, e mostram que o espaço total necessário é $\mathcal{O}(\frac{nc}{p} + c + n)$ com $p \leq n$, ou seja, $\mathcal{O}(\frac{nc}{p} + n)$.

Pode-se observar que quando a capacidade da mochila for maior do que o número de objetos, isto é, $c = \Omega(n)$, este algoritmo gastará mais tempo que o algoritmo de Bellman, não importando o número $p$ de processadores utilizados. Além disso, quando o número de processadores for da mesma ordem que o número de objetos, isto é, $p = \Theta(n)$, a aceleração do algoritmo paralelo torna-se $\mathcal{O}(\frac{n}{c+\log n})$, que é muito ruim quando $c$ é grande. Portanto, este algoritmo somente pode ser prático quando $c$ e $p$ forem muito pequenos.

### III.2.2 Algoritmos de Teng

Os algoritmos apresentados por Teng [Ten90] para a resolução do Problema da Mochila em uma PRAM SIMD EREW baseiam-se numa mesma técnica: a de reduzir o KP para outros problemas já solucionados neste mesmo modelo.

Considerando que $M(m)$ é o número necessário de processadores para multiplicar duas matrizes de ordem $m$, na tabela abaixo estão seus três principais resultados:

| TEMPO | PROCESSADORES |
|---|---|
| $\mathcal{O}(\log^2(nc))$ | $M(c)$ |
| $\mathcal{O}(\log \frac{c}{w_{max}} \log^2 w_{max})$ | $\mathcal{O}(\frac{M(w_{max})}{w_{max} \log \frac{c}{w_{max}}})$ |
| $\mathcal{O}(\log n \log c)$ | $\mathcal{O}(\frac{nc^2}{\log n \log c})$ |

Esses três algoritmos exigem um pré-processamento que consome tempo $\mathcal{O}(\log(nc))$. Além disso, como pode ser observado na tabela, eles não são adaptativos: servem apenas para casos particulares do número de processadores. Principalmente por este motivo, não entraremos em maiores detalhes a respeito deste trabalho. Além disso, se fôssemos descrevê-lo mais a fundo, seria necessário apresentar previamente os outros problemas para os quais o KP foi reduzido. Pensamos que isso não vale a pena, tendo em conta que o próximo algoritmo é considerado pela literatura como o melhor, por ser adaptativo e eficiente.

### III.2.3 Algoritmo de Lin e Storer

O algoritmo de Lin e Storer [LiS90, LiS91] é uma paralelização ótima e adaptativa do algoritmo de Bellman para a PRAM SIMD EREW. Como já dissemos acima, é o melhor resultado conhecido para a resolução do KP01 neste modelo segundo o paradigma da programação dinâmica. No entanto, há um fato curioso com relação a este trabalho: Kindervater e Lenstra [KiL86] já haviam apresentado esta paralelização, que é muito simples e direta. De qualquer forma, os autores também realizaram um estudo sobre a implementação deste algoritmo numa *Connection Machine*, que é uma máquina de topologia hipercubo.

Consideraremos novamente a fórmula de cálculo dos vetores de lucro ótimo $F_j(c) = (f_j(0), f_j(1), \ldots, f_j(c))$ dos subproblemas KNAP$(V_j, c)$, $0 \leq j \leq n$, escrita agora de uma maneira ligeiramente diferente:

$$f_j(i) = \begin{cases} f_{j-1}(i), & 0 \leq i < w_j \\ \max\{f_{j-1}(i), f_{j-1}(i - w_j) + l_j\}, & w_j \leq i \leq c \end{cases}$$

Como se pode observar, o cálculo de cada elemento $f_j(i)$ depende somente de valores de $F_{j-1}(c)$. Portanto, é possível obter-se um paralelização ótima do algoritmo de Bellman calculando-se sucessivamente os $c + 1$ vetores $F_j(c)$ de lucro ótimo.

Considerando uma PRAM SIMD com $p = c + 1$ processadores, cada processador $P_i$, $0 \leq i \leq c$, calcula simultaneamente as soluções $f_j(i)$ para a mochila de capacidade $i$, incrementando passo a passo o número de objetos considerados, ou seja, $j$ variará de 0 a $n - 1$. Portanto, o tempo total deste algoritmo paralelo será proporcional a $n$, que é o número de objetos. No final, $P_i$ armazenará os valores $f_0(i), f_1(i), \ldots, f_{n-1}(i)$, onde $0 \leq i \leq c$, ou seja, os vetores de lucro máximo estarão distribuídos entre todos os processadores, um elemento em cada.

Durante a computação de cada $f_j(i)$, os processadores $P_i$ utilizam um mesmo deslocamento $w_j$ no vetor $f_{j-1}$, o que ocasiona no máximo duas leituras simultâneas numa mesma posição da memória. Estes possíveis conflitos podem ser resolvidos em **tempo constante** no modelo EREW.

A seguir está o algoritmo que calcula os vetores de lucro máximo.

**for** $i \leftarrow 0$ **to** $c$ **do in parallel**
  $f_{-1}[i] \leftarrow 0$
**end for**
**for** $j \leftarrow 0$ **to** $n - 1$ **do**

**for** $i \leftarrow 0$ **to** $c$ **do in parallel**
$f_j[i] \leftarrow f_{j-1}[i]$, $(0 \leq i < w_j)$
$f_j[i] \leftarrow \max\{f_{j-1}[i], f_{j-1}[i - w_j] + l_j\}$, $(w_j \leq i \leq c)$
**end for**
**end for**

Para se encontrar o vetor binário $X$, $f$ deve ser percorrido em todas as suas $n$ posições, sendo que o cálculo de cada $x_i$, $0 \leq i < n$, depende da iteração anterior, o que inviabiliza a paralelização desta fase. No entanto, pode-se utilizar o mesmo algoritmo seqüencial de Bellman, pois seu tempo é também $\mathcal{O}(n)$. Esta é portanto a complexidade do algoritmo paralelo de Lin e Storer para a PRAM SIMD EREW com $p = c + 1$ processadores.

Lin e Storer mostram como tornar adaptativo este algoritmo numa máquina com menos processadores: basta que cada um deles cuide de $q = \lceil \frac{c+1}{p} \rceil$ capacidades da mochila. Em concreto, o processador $P_i$, $0 \leq i < p$, fica com as capacidades que variam entre $iq$ e $(i + 1)q - 1$.

Abaixo está o algoritmo adaptativo que calcula os vetores de lucro ótimo, e que gasta tempo $\mathcal{O}(nc/p)$ e espaço $\mathcal{O}(nc)$, onde $1 \leq p \leq c + 1$:

$q \leftarrow \lceil \frac{c+1}{p} \rceil$
**for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
**for** $k \leftarrow 0$ **to** $q - 1$ **do**
$f_{-1}[iq + k] \leftarrow 0$
**end for**
**end for**
**for** $j \leftarrow 0$ **to** $n - 1$ **do**
**for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
**for** $k \leftarrow 0$ **to** $q - 1$ **do**
$pos \leftarrow iq + k$
$f_j[pos] \leftarrow f_{j-1}[pos]$, $(0 \leq i < \frac{w_j - k}{q})$
$f_j[pos] \leftarrow \max\{f_{j-1}[pos], f_{j-1}[pos - w_j] + l_j\}$, $(\frac{w_j - k}{q} \leq i < p)$
**end for**
**end for**
**end for**

O vetor solução $X$ continua sendo encontrado através do mesmo algoritmo seqüencial

que percorre os vetores de lucro ótimo. Como este algoritmo gasta tempo $\mathcal{O}(n)$, e temos $p \leq c+1$, não há aumento na complexidade final.

A simulação deste algoritmo num hipercubo com o mesmo número $p$ de processadores gasta tempo $\mathcal{O}(\frac{nc}{p}\log p)$, ou seja, tem aceleração $\mathcal{O}(\frac{p}{\log p})$, sendo portanto melhor que o algoritmo de Lee *et al* [LSS88].

### III.2.4 Algoritmos de Goldman e Trystram

Baseando-se também no algoritmo de Bellman, Goldman e Trystram elaboraram dois algoritmos paralelos que resolvem o KP no hipercubo.

É sabido que determinados problemas como o KP podem ser descritos através de grafos de precedência, onde os nós representam tarefas ou instruções e os arcos correspondem às restrições de precedência entre elas. Um processo de paralelização para estes problemas pode ser obtido quando se determina um escalonamento de tarefas, isto é, uma associação de cada uma delas com um tempo de execução e um processador. Entre outras coisas, esta abordagem é muito útil e adequada para a elaboração de algoritmos sistólicos.

O KP já foi bastante estudado em redes sistólicas (caminhos, anéis e grades), pois seu grafo de precedência corresponde a uma grade irregular com $n$ linhas e $c+1$ colunas, onde as arestas horizontais dão "saltos" conforme os pesos dos objetos correspondentes. Na figura III.1 está um exemplo deste grafo. É possível observar que sua construção baseia-se claramente no princípio de otimalidade de Bellman.

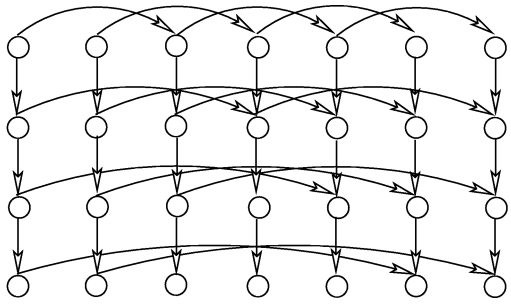

Figura III.1: Grafo de precedência do KP, onde $n = 4$, $c = 6$ e $W = \{2, 3, 4, 5\}$.

A idéia central dos dois trabalhos de Goldman e Trystram é encontrar imersões *(embeddings)* desta grade irregular no grafo do hipercubo e os correspondentes tempos de execução do algoritmo. Para se evitar conflitos de comunicação, os autores empregam a técnica de *pipeline* no envio de dados através dos canais de cada processador.

No primeiro artigo [GoT97], foi possível resolver o problema em tempo $\mathcal{O}(n \log w_{max} +$

$\frac{c}{w_{min}}$) utilizando-se $\mathcal{O}(\frac{cw_{max}}{w_{min}\log w_{max}})$ processadores. No caso em que $p < \lceil\frac{c}{w_{min}}\rceil\lceil\frac{w_{max}}{\log_2 w_{max}}\rceil$, o tempo passa a ser $\mathcal{O}(\frac{nc}{p}\frac{w_{max}}{w_{min}})$, ou seja, obtém-se uma aceleração de $\mathcal{O}(\frac{pw_{min}}{w_{max}})$.

No segundo trabalho [GoT03], Goldman e Trystram obtiveram um resultado ainda melhor: para $w_{min} < p < \frac{c}{\log w_{min}}$ processadores, resolvem o problema em tempo $\mathcal{O}(\frac{nc}{p} + \frac{c}{w_{min}})$, ou seja, conseguiram uma paralelização assintoticamente ótima no hipercubo. No caso em que $p \geq \frac{c}{\log w_{min}}$, a resolução passa a gastar tempo $\mathcal{O}(n\log w_{min} + \frac{c}{w_{min}})$, que, embora seja menor, perde a otimalidade.

Em ambos os trabalhos, os autores explicam como se recupera a solução completa do problema, isto é, os coeficientes de cada peso na solução ótima: isso pode ser feito através do envio de um *array* de *bits* junto com os dados que são transmitidos.

É fácil observar que o espaço necessário em ambas as soluções é $\mathcal{O}(nc)$, da mesma forma que o algoritmo de Bellman.

## III.3 Quadro comparativo dos algoritmos paralelos para a PRAM SIMD

Considerando os resultados apresentados, podemos resumir na tabela abaixo os dados dos algoritmos paralelos que resolvem alguma variante do Problema da Mochila nos diferentes modelos da PRAM SIMD.

Com exceção do algoritmo de Lin e Storer, que resolve o KP01 através do paradigma da programação dinâmica, todos os outros baseiam-se no paradigma das listas e resolvem o SSPd. Para os algoritmos de Chang *et al.* e Lou e Chang não indicamos os resultados publicados pelos autores, mas as suas respectivas correções (*cfr.* [SSY02]).

| ALGORITMO | VARIANTE | MODELO | TEMPO | ESPAÇO | PROCESSADORES | ACELERAÇÃO |
|---|---|---|---|---|---|---|
| Karnin | SSPd | EREW | $\mathcal{O}(n2^{n/2})$ | $\mathcal{O}(2^{n/6})$ | $p = 2.2^{n/6}$ | $\mathcal{O}(1/\log p)$ |
| Ferreira | SSPd | CREW | $\mathcal{O}(n2^{n/2}/p)$ | $\mathcal{O}(2^{n/2})$ | $\mathcal{O}(1) < p < \mathcal{O}(2^{n/2})$ | $\mathcal{O}(p/n)$ |
| Ferreira e Robson | SSPd | EREW | $\mathcal{O}(n2^{n/2}/p)$ | $\mathcal{O}(2^{n/4})$ | $1 \leq p \leq 2^{n/4}$ | $\mathcal{O}(p/n)$ |
| Chang *et al.* | SSPd | CREW | $\mathcal{O}(n2^{n/2})$† | $\mathcal{O}(2^{n/2})$† | $p = \mathcal{O}(2^{n/8})$ | $\mathcal{O}(1/\log p)$ |
| Lou e Chang | SSPd | CREW | $\mathcal{O}(n2^{n/2})$† | $\mathcal{O}(2^{n/2})$† | $p = \mathcal{O}(2^{n/8})$ | $\mathcal{O}(1/\log p)$ |
| Lin e Storer | KP01 | EREW | $\mathcal{O}(nc/p)$ | $\mathcal{O}(nc)$ | $1 \leq p \leq c+1$ | $\mathcal{O}(p)$‡ |

† Valores corrigidos.
‡ Aceleração calculada em relação ao algoritmo de Bellman.

Nas paralelizações desenvolvidas através do paradigma das listas, a aceleração refere-se ao tempo do *algoritmo das duas listas* [HoS74], que é o melhor resultado seqüencial. Pode-se observar que nenhuma deles atinge a aceleração ótima $\mathcal{O}(p)$.

O algoritmo de Lin e Storer atinge a aceleração ótima na resolução do KP01 numa PRAM SIMD EREW, uma vez que o melhor algoritmo seqüencial por programação dinâmica para este problema continua sendo o de Bellman [Bel57]. No entanto, como

explicamos no capítulo anterior, há resoluções seqüenciais de variantes do Problema da Mochila que gastam tempo $o(nc)$: são os casos, por exemplo, dos algoritmos de Yanasse e Soma [YaS87] e de Soma e Toth [SoT02], ambos voltados para o SSP. Por isso, podemos dizer que o algoritmo de Lin e Storer não é uma paralelização ótima para a resolução do SSP.

Tendo isto em conta, abriram-se dois campos de pesquisa para obtermos resultados originais na resolução do Problema da Mochila numa PRAM SIMD: o desenvolvimento de uma paralelização que seja ótima e adaptativa segundo o paradigma das listas, e a obtenção de algoritmos paralelos para o SSP que sejam melhores do que o de Lin e Storer. Estes resultados serão apresentados nos dois próximos capítulos.

# Capítulo IV

# Uma paralelização ótima e adaptativa para o *algoritmo das duas listas*

Neste capítulo apresentamos um algoritmo paralelo original que resolve o SSP numa PRAM SIMD CREW através do paradigma das listas. Sua inspiração surgiu a partir do trabalho de Lou e Chang [LoC97].

A princípio, não parece difícil tornar adaptativas as fases de *descarte* e de *busca* explicadas na subseção III.1.5. Utilizando-se $p = 2^q$ processadores, $0 \leq q < n/2$, as listas ordenadas $A$ e $B$ poderiam ser divididas em $2^q$ blocos. Desse modo, a *fase de busca* gastaria tempo $\mathcal{O}(2^{n/2-q})$, pois é proporcional ao tamanho dos blocos, e poderia ser utilizado o algoritmo seqüencial que resolve o SSP ao invés do SSPd (*cfr.* subseção II.2.1). Além disso, a *fase de descarte* geraria no máximo $2.2^q$ pares de blocos em tempo $\mathcal{O}(2^q)$. No entanto, se fizermos desse modo, esta última fase não seria ótima: para $n/4 < q < n/2$, o fator $2^q$ é dominante sobre $2^{n/2-q}$.

Portanto, para obtermos um algoritmo adaptativo ótimo para o Problema da Mochila, todas as suas fases precisam gastar tempo $\mathcal{O}(2^{n/2-q})$. Por isso, torna-se necessário elaborar algoritmos para as fases de *geração* e de *descarte* que não ultrapassem este limite.

A seguir, apresentaremos algoritmos adaptativos para as três fases em que este tempo ótimo não é ultrapassado. Este novo algoritmo paralelo é mais eficiente e completo que o de Lou e Chang: é adaptativo, conta com uma *fase de geração* correta, faz a alocação dos pares de blocos na *fase de busca* e resolve o SSP ao invés do SSPd. Além disso, é o único em toda a literatura a alcançar na PRAM SIMD uma aceleração ótima com relação ao *algoritmo das duas listas*.

## IV.1 *Fase de geração*

Descreveremos abaixo a geração da lista $A$ em ordem não-decrescente, que corresponde a todas as possíveis somas da primeira metade dos pesos $\{w_0, w_1, \ldots, w_{(n-1)/2}\}$. O número total de processadores disponíveis é $p = 2^q$, onde $0 \leq q \leq \frac{n}{2} - 2\log_2 n$. A geração de $B$ em ordem não-crescente a partir dos outros $n/2$ pesos é feita de modo análogo.

**Passo 1:** Gere em uma lista $A$ inicialmente vazia todas as possíveis somas dos primeiros $q$ pesos, isto é, de $\{w_0, w_1, \ldots, w_{q-1}\}$. Para isso, cada processador $P_i$, $0 \leq i < 2^q$, gera a soma dos pesos $w$'s correspondentes ao seu índice expresso em código binário.

**Passo 2:** Ordene a lista $A$ crescentemente, utilizando os $p = 2^q$ processadores.

**Passo 3:**

**for** $i \leftarrow q$ **to** $(n-1)/2$ **do in parallel**

**(3.1)** Copie a lista $A$ em outra lista $A'$ e
adicione $w_i$ a cada elemento de $A'$.

**(3.2)** Faça uma intercalação paralela entre as listas $A$ e $A'$,
armazenando a lista resultante em $A$.

**end for**

O **Passo 1** por ser feito claramente em tempo $\mathcal{O}(q)$. No **Passo 2**, podemos utilizar um algoritmo de tempo $\mathcal{O}(q)$ [Col88]. No entanto, como se verá logo adiante, outros algoritmos paralelos de ordenação de tempo $\mathcal{O}(q^2)$ (como [Bat68], por exemplo) também poderiam ser utilizados, pois isso não afetará a complexidade de tempo desta fase.

A análise da complexidade do **Passo 3** torna-se mais fácil considerando separadamente as iterações dos comandos **(3.1)** e **(3.2)**. A primeira iteração de **(3.1)** é executada em tempo constante, pois temos $p = 2^q$ processadores e uma lista com $2^q$ elementos: cada processador $P_j$ ficaria com a incumbência de somar o valor de $w_i$ ao $j$-ésimo elemento da lista $A'$. No passo seguinte, este tempo será o dobro, pois a lista terá dobrado de tamanho; e assim por diante, até o último passo. Portanto, o tempo total gasto pelo comando **(3.1)** é $\sum_{k=0}^{(n-1)/2-q} \mathcal{O}(2^k) = \mathcal{O}(2^{n/2-q})$.

Por outro lado, é sabido que, em uma máquina PRAM SIMD CREW com $p$ processadores, duas listas ordenadas de tamanho $k$ podem ser intercaladas em tempo $\mathcal{O}(k/p+\log k)$ (veja, por exemplo, [Akl89]). Além disso, é possível observar que o comando **(3.2)** é executado $(n-1)/2-q+1$ vezes e, para $p = 2^q$ processadores, é imediato concluir que o tempo

gasto pelas iterações deste comando no **Passo 3** será $\mathcal{O}(2^{n/2-q}+n^2) = O(2^{n/2-q}+2^{2\log_2 n})$. Portanto, quando $0 \leq q \leq \frac{n}{2} - 2\log_2 n$, este tempo é $\mathcal{O}(2^{n/2-q})$.

Dessa forma, o **Passo 3** tem complexidade de tempo $\mathcal{O}(2^{n/2-q})$, que é a complexidade final da *fase de geração*.

## IV.2 *Fase de descarte*

A nova *fase de descarte* resume-se em três passos principais:

(i) Buscas binárias simultâneas na lista $B$ para gerar na memória compartilhada uma lista de $p$ quádruplas;

(ii) Preenchimento do quarto campo dessas quádruplas;

(iii) Buscas binárias simultâneas nessa lista de quádruplas com o objetivo de alocar no máximo dois pares de blocos para cada processador.

Vamos explicar detalhadamente cada um desses passos. No primeiro deles, de modo análogo ao algoritmo de Lou e Chang, cada bloco $A_i$, com $e = 2^{n/2-q}$ elementos, é alocado ao processador $P_i$, para $0 \leq i < 2^q$. Baseados no *Lema 1* e no *Lema 2* (subseção III.1.5), cada processador executa duas buscas binárias na lista $B$ para encontrar os índices $j$ e $j + t - 1$ que identificam os $t$ blocos consecutivos $B_j$, $B_{j+1}$, ..., $B_{j+t-1}$ que serão associados ao bloco $A_i$. Com esta informação, o processador $P_i$ escreve a quádrupla $(i, j, t, 0)$ no $i$-ésimo elemento de um vetor $Q$ quadri-dimensional de $p$ posições presente na memória compartilhada. Se nenhum par de blocos for associado a $A_i$, então $P_i$ escreverá a quádrupla $(i, \#, 0, 0)$, onde $\#$ é um valor *dummy*. É possível observar que os acessos aos elementos da lista $B$ são executados concorrentemente, ou seja, os processadores trabalham em paralelo.

Este passo é apresentado a seguir. $Q_i$ é a $i$-ésima quádrupla (ou elemento) de $Q$, $0 \leq i < 2^q$, e $Q_i[y]$ é o $y$-ésimo campo dessa quádrupla, $1 \leq y \leq 4$. Evidentemente, se a solução ótima for encontrada casualmente durante este passo, todo o processamento será interrompido.

*GeraçãodasQuádruplas*
$l \leftarrow 0$
$r \leftarrow 2^q - 1$
**while** $l < r$ **do**

$m \leftarrow \lfloor (l+r)/2 \rfloor$
**if** $A_i[0] + B_m[e-1] = c$ **then** solução encontrada **end if**
**if** $A_i[0] + B_m[e-1] > c$ **then** $l \leftarrow m$
**else** $r \leftarrow m$
**end if**
**end while**
**if** $A_i[0] + B_l[e-1] > c$ **then** $Q_i \leftarrow (i, \#, 0, 0)$
**else**
$j \leftarrow l$
$r \leftarrow 2^q - 1$
**while** $l < r$ **do**
$m \leftarrow \lfloor (l+r)/2 \rfloor$
**if** $A_i[e-1] + B_m[0] = c$ **then** solução encontrada **end if**
**if** $A_i[e-1] + B_m[0] < c$ **then** $r \leftarrow m$
**else** $l \leftarrow m$
**end if**
**end while**
$Q_i \leftarrow (i, j, l-j+1, 0)$
**end if**

A primeira busca binária procura o bloco de $B$ mais à esquerda (ou seja, com maiores valores) tal que a soma de seus elementos com os elementos de $A_i$ não seja maior do que $c$. A segunda busca faz algo análogo com relação ao bloco de $B$ mais à direita (com menores valores). É imediato concluir que este passo pode ser executado em tempo $\mathcal{O}(\log 2^q) = \mathcal{O}(q)$.

A figura IV.1 representa a busca binária realizada pelo processador $P_i$. As setas indicam a ordenação dos elementos das listas $A$ e $B$.

É importante ressaltar que a lista $Q$ na memória compartilhada terá exatamente $p$ quádruplas, embora represente um número máximo de $2p$ pares de blocos (cfr. *Lema 4*). Para associar no máximo dois pares de blocos $(A_i, B_j)$ a cada um dos $p$ processadores — esta importante tarefa nem foi mencionada em [LoC97] —, se utilizará o quarto campo das quádruplas, cuja função será indicar o *ranking* do primeiro par representado pela quádrupla a qual pertence. Dessa forma, cada quádrupla $(i, j, t, k)$ representará $t$ pares de blocos — de $(A_i, B_j)$ até $(A_i, B_{j+t-1})$ —, sendo que, dentre todos os pares selecionados em $Q$, há exatamente $k$ pares antes de $(A_i, B_j)$. Assim, temos que $Q_j[4] = \sum_{i=0}^{j-1} Q_i[3]$.

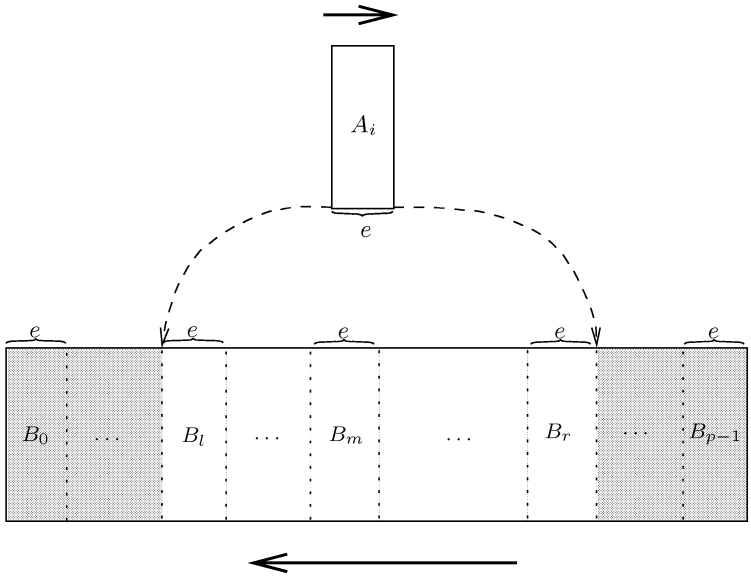


Figura IV.1: Busca binária no vetor $B$ a partir de $P_i$.

Algoritmos paralelos para cálculo de *ranking* são bastante conhecidos na literatura [NaS81, NaS82]. Indicamos a seguir o algoritmo executado por cada processador $P_i$, $0 \leq i < p$, que preenche o quarto campo das quádruplas segundo a definição acima. $E$ é um vetor auxiliar de $p$ posições também presente na memória compartilhada.

*PreencheRanking*
$E[i] \leftarrow Q_i[3]$
**for** $j \leftarrow 0$ **to** $q-1$ **do**
    **if** $\lfloor i/2^j \rfloor \bmod 2 = 1$ **then** TotalVizinho $\leftarrow E[i-2^j]$
                **else** TotalVizinho $\leftarrow E[i+2^j]$
    **end if**
    **if** $\lfloor i/2^j \rfloor \bmod 2 = 1$ **then** $Q_i[4] \leftarrow Q_i[4] +$ TotalVizinho **end if**
    $E[i] \leftarrow E[i] +$ TotalVizinho
**end for**

Inicialmente, há $p = 2^q$ grupos, cada um com uma única quádrupla. A cada iteração, dois grupos vizinhos são combinados em um novo grupo. A quantidade total de pares representados no grupo ao qual pertence a quádrupla $i$ é armazenado em $E[i]$. No final, haverá um único grupo com todas as $p$ quádruplas, com $E[i] = \sum_{j=0}^{p-1} Q_j[3]$, $0 \leq i < p$. É fácil ver que este algoritmo também gasta tempo $\mathcal{O}(q)$

Assim que o quarto campo das quádruplas tenha sido preenchido, é necessário determinar e alocar no máximo dois pares de blocos para cada um dos $p = 2^q$ processadores.

A função *EncontrePar* descrita abaixo, através de uma busca binária em $Q$, retorna o $y$-ésimo par representado pelas quádruplas, onde $0 \leq y < 2p$.

*EncontrePar* $(y)$
$l \leftarrow 0$
$r \leftarrow 2^q - 1$
**while** $l < r$ **do**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    **if** $(Q_m[3] \neq 0)$ **and** $(y \geq Q_m[4])$ **and** $(y < Q_m[4] + Q_m[3])$
      **then return** $(Q_m[1], y + Q_m[2] - Q_m[4])$
      **else**
        **if** $y < Q_m[4]$ **then** $r \leftarrow m - 1$
                **else** $l \leftarrow m + 1$
        **end if**
    **end if**
**end while**
**return** $(\infty, \infty)$

Cada processador $P_i$ fará duas chamadas para esta função, com parâmetros $i$ e $i + p$, obtendo índices de no máximo dois pares de blocos nos quais deverá realizar a *fase de busca*. A função *EncontrePar* gasta tempo $\mathcal{O}(q)$.

Portanto, o corpo principal da *fase de descarte* será:

*GeraçãodasQuádruplas*
*PreencheRanking*
$(i_1, j_1) \leftarrow$ *EncontrePar* $(i)$
$(i_2, j_2) \leftarrow$ *EncontrePar* $(i + p)$

Com isso, o tempo total desta fase será $\mathcal{O}(q)$.

Vamos apresentar um exemplo de como ela funciona para o caso em que $q = 3$, ou seja, quando dispomos de $p = 8$ processadores. Cada processador $P_i$, depois de fazer os testes necessários, determinou os blocos de $B$ que devem ser associados ao bloco $A_i$. Suponhamos que essa associação tenha sido a seguinte: para o bloco $A_0$, interessam apenas os blocos $B_0$ a $B_2$; para $A_1$, os blocos $B_2$ e $B_3$; para $A_3$, o bloco $B_4$; para $A_4$, os blocos $B_5$ e $B_6$; para $A_5$, o bloco $B_6$; e para $A_7$, o bloco $B_7$. Para $A_2$ e $A_6$, não há blocos de $B$ associados. Portanto, $Q$ corresponderá às seguintes quádruplas: $(0, 0, 3, 0)$, $(1, 2, 2, 0)$,

$(2, \#, 0, 0)$, $(3, 4, 1, 0)$, $(4, 5, 2, 0)$, $(5, 6, 1, 0)$, $(6, \#, 0, 0)$, $(7, 7, 1, 0)$. Após o cálculo dos *rankings*, as quádruplas passarão a ser $(0, 0, 3, 0)$, $(1, 2, 2, 3)$, $(2, \#, 0, 5)$, $(3, 4, 1, 5)$, $(4, 5, 2, 6)$, $(5, 6, 1, 8)$, $(6, \#, 0, 9)$, $(7, 7, 1, 9)$. Dessa forma, os processadores podem encontrar os pares de blocos nos quais deverão realizar a *fase de busca*. Neste exemplo, como foram selecionados 10 pares de blocos, somente os dois primeiros processadores receberão dois pares, enquanto que os outros ficarão com apenas um. O processador $P_1$, por exemplo, ao procurar o segundo e o décimo pares de blocos selecionados, encontrará $(A_0, B_1)$ e $(A_7, B_7)$, respectivamente.

## IV.3 *Fase de busca*

Nesta última fase, cada processador executa o algoritmo seqüencial da *fase de busca* no par de blocos $(A_{i_1}, B_{j_1})$, desde que $i_1 < p$ e $j_1 < p$, e também no par $(A_{i_2}, B_{j_2})$, desde que $i_2 < p$ e $j_2 < p$. Assim que encontrar a melhor solução correspondente a eles, irá escrevê-la na $i$-ésima posição de um outro vetor de tamanho $p$ presente na memória compartilhada. No pior caso, cada processador $P_i$ fará duas buscas entre um máximo de $2^{n/2-q}$ valores, gastando tempo $\mathcal{O}(2^{n/2-q})$.

Este novo vetor conterá $p$ soluções parciais, e a melhor entre elas será a solução ótima do SSP. Os $p$ processadores, trabalhando em paralelo, podem encontrar facilmente esse máximo valor em tempo $\mathcal{O}(\log p) = \mathcal{O}(q)$.

Dessa forma, a complexidade final de tempo desta *fase de busca* é $\mathcal{O}(2^{n/2-q})$.

## IV.4 Comparações com os outros algoritmos paralelos

As complexidades de tempo das três fases do novo algoritmo e de seus respectivos passos intermediários estão indicadas na tabela abaixo. A quantidade de processadores envolvidos é $p = 2^q$, onde $0 \leq q \leq \frac{n}{2} - 2\log_2 n$.

| FASES | PASSOS INTERMEDIÁRIOS | TEMPO |
|---|---|---|
| *Geração* | **Passo 1** | $\mathcal{O}(q)$ |
| | **Passo 2** | $\mathcal{O}(q)$ |
| | **Passo 3** | $\mathcal{O}(2^{n/2-q})$ |
| *Descarte* | *GeraçãodasQuádruplas* | $\mathcal{O}(q)$ |
| | *PreencheRanking* | $\mathcal{O}(q)$ |
| | *EncontrePar* | $\mathcal{O}(q)$ |
| *Busca* | Buscas seqüenciais nos pares de blocos | $\mathcal{O}(2^{n/2-q})$ |
| | Máximo valor entre as soluções parciais | $\mathcal{O}(q)$ |

Como já dissemos, este é o primeiro algoritmo paralelo adaptativo com aceleração ótima para o Problema da Mochila segundo o paradigma das listas, elaborado para o modelo PRAM SIMD CREW com $p = 2^q$ processadores, onde $0 \leq q \leq \frac{n}{2} - 2\log_2 n$. Ele é composto por três fases: (i) *geração*, (ii) *descarte* e (iii) *busca*. As fases de *geração* e de *busca* têm complexidade $\mathcal{O}(2^{n/2-q})$, enquanto que a *fase de descarte* gasta tempo $\mathcal{O}(q)$. Portanto, a complexidade de tempo do novo algoritmo é $\mathcal{O}(2^{n/2-q})$, enquanto a de espaço é $\mathcal{O}(2^{n/2})$.

A seguir, comparamos as complexidades de tempo e espaço dos principais algoritmos paralelos que resolvem variantes do Problema da Mochila em diferentes modelos da PRAM SIMD através do paradigma das listas. Também indicamos o número de processadores utilizados e as respectivas acelerações obtidas, todas elas em relação ao *algoritmo das duas listas*.

| ALGORITMO | VARIANTE | MODELO | TEMPO | ESPAÇO | PROCESSADORES | ACELERAÇÃO |
|---|---|---|---|---|---|---|
| Karnin | SSPd | EREW | $\mathcal{O}(n2^{n/2})$ | $\mathcal{O}(2^{n/6})$ | $p = 2.2^{n/6}$ | $\mathcal{O}(1/n)$ |
| Ferreira | SSPd | CREW | $\mathcal{O}(n2^{n/2}/p)$ | $\mathcal{O}(2^{n/2})$ | $\mathcal{O}(1) < p < \mathcal{O}(2^{n/2})$ | $\mathcal{O}(p/n)$ |
| Ferreira e Robson | SSPd | EREW | $\mathcal{O}(n2^{n/2}/p)$ | $\mathcal{O}(2^{n/4})$ | $1 \leq p \leq 2^{n/4}$ | $\mathcal{O}(p/n)$ |
| Lou e Chang | SSPd | CREW | $\mathcal{O}(n2^{n/2})$† | $\mathcal{O}(2^{n/2})$† | $p = \mathcal{O}(2^{n/8})$ | $\mathcal{O}(1/\log p)$ |
| **Novo algoritmo** | SSP | CREW | $\mathcal{O}(2^{n/2}/p)$ | $\mathcal{O}(2^{n/2})$ | $1 \leq p \leq 2^{n/2}/n^2$ | $\mathcal{O}(p)$ |

† Valores corrigidos [SSY02].

Há uma clara evidência na melhora obtida pelo nosso algoritmo, tanto com relação à aceleração como na adaptabilidade do número de processadores. Este resultado já foi anunciado em uma conferência internacional [YSS01] e deverá ser publicado futuramente [SSY03].

# Capítulo V

# Paralelizações em tempo $o(nc/p)$ e espaço $\mathcal{O}(n+c)$

Apresentaremos neste capítulo três paralelizações adaptativas originais para a solução do SSP na PRAM SIMD CREW que se baseiam no paradigma da programação dinâmica: as duas primeiras são do algoritmo de Yanasse e Soma [YaS87], e a outra corresponde ao algoritmo de Soma e Toth [SoT02]. Como todas gastam tempo $o(nc/p)$ e espaço $\mathcal{O}(n+c)$, são melhores que os resultados obtidos pelo algoritmo de Lin e Storer [LiS90, LiS91].

## V.1 Algoritmo 1: com $p \leq w_{min}$ processadores

Considerando o vetor $g$ utilizado por Yanasse e Soma [YaS87], basta dividir as suas $c-2w_{min}$ posições em blocos de tamanho $p$. Em cada passo deste algoritmo paralelo, cada processador calcula uma única posição do bloco corrente: para isso, testa seqüencialmente todos os $n$ pesos, mas utiliza apenas valores de $g$ já calculados, isto é, que estejam em posições inferiores do vetor. Para que isso possa ser feito em paralelo, é necessário permitir a leitura simultânea do vetor $g$ e exigir que $p \leq w_{min}$.

Cada processador $P_i$, $0 \leq i < p$, terá sua solução parcial $opt_i$, que será a melhor dentre as que ele calculou. No final, a solução ótima do SSP será $\max\{opt_i\}$, $0 \leq i < p$, que pode ser encontrada em paralelo pelos $p$ processadores em tempo $\mathcal{O}(\log p)$.

A descrição deste algoritmo está logo abaixo. Vale a pena comentar que o vetor $g$ precisará de $d = w_{max} - w_{min} - 1$ posições extras na sua inicialização (todas elas anteriores à posição 0), para assim tornar possível os acessos realizados durante o processamento do primeiro bloco. Este algoritmo está representado na figura V.1, que mostra o cálculo da posição $g[pos]$ do bloco $k$.

$d \leftarrow w_{max} - w_{min} - 1$

**for** $k \leftarrow 0$ **to** $\lceil (c+d)/p \rceil - 1$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        $g[kp+i-d] \leftarrow n$
    **end for**
**end for**
$opt_i \leftarrow 0$
**for** $k \leftarrow \lceil n/p \rceil - 1$ **downto** $0$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        $g[w_{kp+i}] \leftarrow kp+i$
        $opt_i \leftarrow \max\{opt_i, w_{kp+i}\}$
    **end for**
**end for**
**for** $k \leftarrow 0$ **to** $\lceil (c-2w_{min})/p \rceil - 1$ **do**
    **for** $i \leftarrow 1$ **to** $p$ **do in parallel**
        $pos \leftarrow kp+i+w_{min}$
        **for** $j \leftarrow n-1$ **downto** $0$ **do**
            **if** $(g[pos-w_j] < j)$ **and** $(g[pos] > j)$ **then**
                $g[pos] \leftarrow j$
                $opt_i \leftarrow \max\{opt_i, pos\}$
            **end if**
        **end for**
    **end for**
**end for**

A solução ótima pode ser encontrada em tempo $\mathcal{O}(\log p)$, e a recuperação do vetor binário $X$ em tempo $\mathcal{O}(n)$ através do algoritmo seqüencial. Portanto, o tempo total será $\mathcal{O}(\frac{n}{p}(c-2w_{min}) + \frac{c+w_{max}-w_{min}}{p} + n + \log p)$. Como $\mathcal{O}(\log p) = \mathcal{O}(\log w_{min})$, na grande maioria dos casos práticos este tempo pode ser considerado como $o(nc/p)$. O espaço necessário é $\mathcal{O}(n+c)$.

## V.2 Algoritmo 2: com $p \leq n$ processadores

Vamos descrever inicialmente o caso em que $p = n$, para depois generalizá-lo. Neste caso, em cada passo do *loop* principal os cálculos se baseiam numa única posição do vetor $g$: cada processador $P_i$, $0 \leq i < n$, fica encarregado de testar o peso $w_i$ com o valor desta

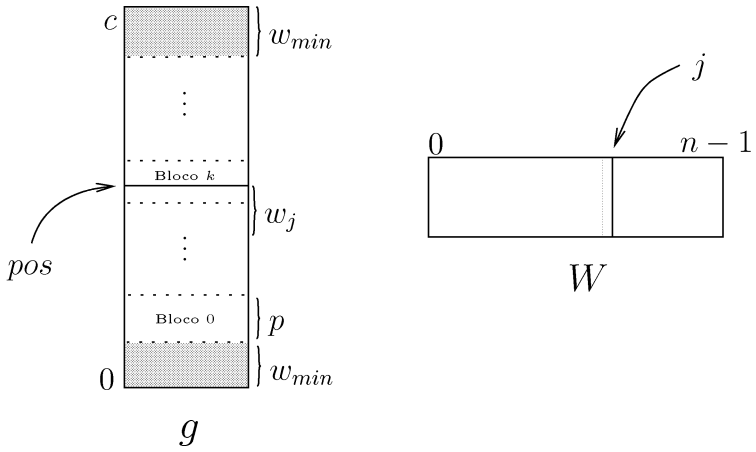


Figura V.1: Cálculo do vetor $g$ no Algoritmo 1.

posição.

Isto gera um problema: as possíveis tentativas de escrita simultânea em $g$, que ocorrerão quando houver pelo menos dois pesos de mesmo valor. Este problema pode ser evitado com uma ordenação prévia dos pesos e com o posterior preenchimento de um vetor *enabled* de dimensão $n$: sua $i$-ésima posição indicará a existência ou não de outro peso de índice menor com valor igual a $w_i$, $0 \leq i < n$. Com esta informação, cada processador terá condições de saber se é sua vez de escrever em $g$, evitando assim os conflitos.

Abaixo está o algoritmo executado pelo processador $P_i$. Na chamada da função *OrdenaçãoParalela*, o vetor $W$ é passado por referência, ou seja, os pesos são trocados de tal forma a ficarem em ordem não-decrescente. A figura V.2 representa o modo como o algoritmo calcula $g[j]$; neste momento, o vetor $W$ já foi ordenado.

**for** $k \leftarrow 0$ **to** $\lceil c/p \rceil - 1$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        $g[kp+i] \leftarrow n$
    **end for**
**end for**
*OrdenaçãoParalela*$(W)$
*enabled*$[i] \leftarrow$ **true**, (i=0)
**if** $w_i = w_{i-1}$, $(i > 0)$
  **then** *enabled*$[i] \leftarrow$ **false**
  **else** *enabled*$[i] \leftarrow$ **true**
**end if**
$opt_i \leftarrow 0$

**for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
    **if** $enabled[i]$ **then**
        $g[w_i] \leftarrow i$
        $opt_i \leftarrow \max\{opt_i, w_i\}$
    **end if**
**end for**
**for** $j \leftarrow w_{min}$ **to** $c - w_{min}$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        **if** $(i = g[j]+1)$ **or** $(i > g[j]$ **and** $enabled[i])$ **then**
            $w' \leftarrow j + w_i$
            **if** $w' \leq c$ **then**
                $g[w'] \leftarrow \min\{g[w'], i\}$
                $opt_i \leftarrow \max\{opt_i, w'\}$
            **end if**
        **end if**
    **end for**
**end for**

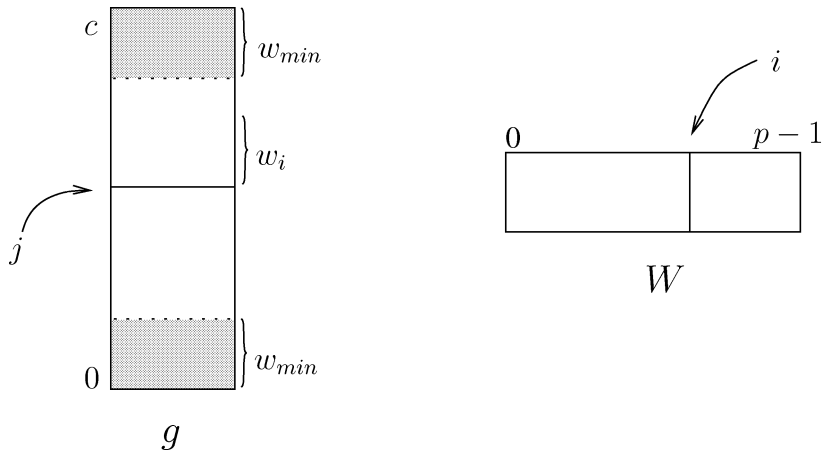


Figura V.2: Cálculo do vetor $g$ no Algoritmo 2, quando $p = n$.

Considerando que a ordenação paralela dos $n$ pesos utilizando $n$ processadores pode ser realizada em tempo $\mathcal{O}(\log n)$ [Col88] ou $\mathcal{O}(\log^2 n)$ [Bat68], que o cálculo da solução ótima é feito em tempo $\mathcal{O}(\log p) = \mathcal{O}(\log n)$, e que a recuperação do vetor $X$ pode ser realizada em tempo $\mathcal{O}(n)$ através do algoritmo seqüencial, o tempo total gasto por este algoritmo paralelo é $\mathcal{O}((c-2w_{min}) + \frac{c}{n} + n)$. O espaço necessário continua sendo $\mathcal{O}(n+c)$.

Este algoritmo pode se tornar adaptativo para o caso em que $p \leq n$: basta que cada processador $P_i$, $0 \leq i < p$, fique responsável pelos testes de $q = \lceil n/p \rceil$ pesos na fase do cálculo de $g$; concretamente, pelos pesos $w_j$ onde $iq \leq j < (i+1)q$. O vetor *enabled* também deverá ter dimensão $p$: sua $i$-ésima posição, $0 \leq i < p$, será **false** quando o primeiro peso de $P_i$ for igual ao último de $P_{i-1}$, isto é, quando $w_{iq} = w_{iq-1}$, e **true** em caso contrário.

A descrição deste algoritmo paralelo adaptativo está logo abaixo.

**for** $k \leftarrow 0$ **to** $\lceil c/p \rceil - 1$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        $g[kp+i] \leftarrow n$
    **end for**
**end for**
*OrdenaçãoParalela*$(W)$
$q \leftarrow \lceil n/p \rceil$
*enabled*$[i] \leftarrow$ **true**, (i=0)
**if** $w_{iq} = w_{iq-1}$, $(i > 0)$
  **then** *enabled*$[i] \leftarrow$ **false**
  **else** *enabled*$[i] \leftarrow$ **true**
**end if**
$opt_i \leftarrow 0$
**for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
    **for** $k \leftarrow q-1$ **downto** $0$ **do**
        $g[w_{iq+k}] \leftarrow iq+k$
        $opt_i \leftarrow \max\{opt_i, w_{iq+k}\}$
    **end for**
**end for**
**for** $j \leftarrow w_{min}$ **to** $c - w_{min}$ **do**
    **for** $i \leftarrow 0$ **to** $p-1$ **do in parallel**
        **for** $k \leftarrow 0$ **to** $q-1$ **do**
            **if** $(iq+k = g[j]+1)$ **or**
              $((iq+k > g[j])$ **and** $((w_{iq+k} > w_{iq})$ **or** $(w_{iq+k} = w_{iq}$ **and** *enabled*$[i]))$ **then**
                $w' \leftarrow j + w_{iq+k}$
                **if** $w' \leq c$ **then**
                    $g[w'] \leftarrow \min\{g[w'], iq+k\}$

$opt_i \leftarrow \max\{opt_i, w'\}$

**end if**

**end if**

**end for**

**end for**

**end for**

Na figura V.3, que mostra o algoritmo calculando uma posição de $g$, é possível observar também a divisão do vetor $W$ em $p$ blocos, após ter sido ordenado.

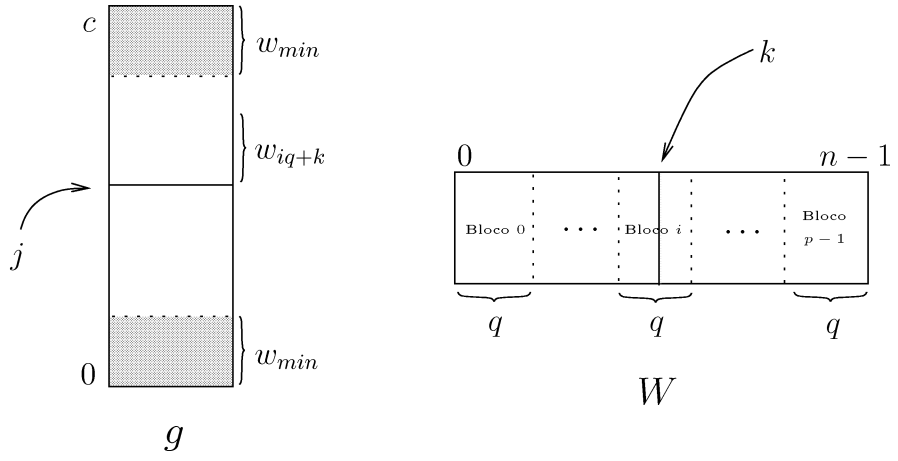


Figura V.3: Cálculo do vetor $g$ no Algoritmo 2.

Como a ordenação paralela dos $n$ pesos utilizando $p \leq n$ processadores em uma PRAM SIMD CREW pode ser realizada em tempo $\mathcal{O}(\frac{n}{p}\log n + \log p \log n)$ [Akl89], o cálculo da solução ótima continua exigindo tempo $\mathcal{O}(\log n)$, e a recuperação do vetor $X$ pode ser feita em tempo $\mathcal{O}(n)$, o tempo total gasto por este algoritmo paralelo adaptativo é $\mathcal{O}(\frac{n}{p}(c - 2w_{min}) + \frac{c}{p} + \frac{n}{p}\log n + \log p \log n + n)$. O espaço necessário permanece $\mathcal{O}(n + c)$.

Podemos analisar esta complexidade de tempo dividindo-a em dois intervalos: quando $p \geq \log n$, será $\mathcal{O}(\frac{n}{p}(c - 2w_{min}) + \frac{c}{p} + n)$. Neste primeiro caso, que chamaremos de **Algoritmo** $2a$, a complexidade de tempo é claramente $o(nc/p)$. Por outro lado, quando $p \leq \log n$, a complexidade será $\mathcal{O}(\frac{n}{p}(c - 2w_{min} + \log n) + \frac{c}{p})$: corresponderá ao **Algoritmo** $2b$.

É fácil observar que o algoritmo apresentado inicialmente, em que $p = n$, é um caso particular do Algoritmo $2a$.

## V.3 Algoritmo 3: com $\log_2(n - 2\log_2 c) \leq p \leq n - 2\log_2 c$ processadores

Apresentaremos agora uma paralelização do algoritmo seqüencial de Soma e Toth [SoT02]. Como este algoritmo se baseia também no paradigma das listas, utilizaremos alguns resultados da paralelização ótima e adaptativa do capítulo anterior. Por isso, será preciso considerar que o número de processadores da PRAM SIMD CREW é $p = 2^q$, onde $\log_2 \log_2(n - 2\log_2 c) \leq q \leq \log_2(n - 2\log_2 c)$. Este intervalo no número de processadores se deve ao fato de que também utilizamos nesta paralelização o Algoritmo $2a$, como se verá adiante.

De modo análogo ao algoritmo seqüencial de Soma e Toth, o Algoritmo 3 também terá quatro fases, que serão descritas a seguir.

***Fase 1:*** **Geração de $g_1$ com as combinações de $\log_2 c$ pesos.** Nesta fase, utilizaremos a mesma idéia descrita na *fase de geração* do algoritmo do capítulo anterior. Sem perda de generalidade, chamaremos esses $\log_2 c$ pesos de $w_0, w_1, \ldots, w_{\log_2 c - 1}$. Esta fase é composta por quatro passos, que serão descritos a seguir. Inicialmente, será gerada uma lista $L$ de tamanho $c$ que conterá todas as possíveis somas desses $\log_2 c$ pesos em ordem não-decrescente.

**Passo 1:** Gere em uma lista $L$ inicialmente vazia todas as possíveis somas entre os pesos $w_0, w_1, \ldots, w_{q-1}$. Para isso, cada processador $P_i$, $0 \leq i < 2^q$, gera a soma dos pesos $w$'s correspondentes ao seu índice expresso em código binário. Além disso, para cada soma gerada, é guardado o índice *ind* do maior peso que participa nesta soma.

**Passo 2:** Ordene a lista $L$ crescentemente, utilizando os $2^q$ processadores. Quando houver somas de mesmo valor, considere o valor de *ind* como critério de desempate.

**Passo 3:**

**for** $i \leftarrow q$ **to** $\log_2 c - 1$ **do in parallel**

**(3.1)** Copie a lista $L$ em outra lista $L'$ e adicione $w_i$ a cada elemento de $L'$, calculando também seus novos índices *ind*.

**(3.2)** Faça uma intercalação paralela entre as listas $L$ e $L'$, armazenando a lista resultante em $L$.

**end for**

Nas intercalações acima, é seguido o mesmo critério de desempate da ordenação, ou seja, utiliza-se o valor de $ind$ quando houver somas iguais.

**Passo 4:** Seja $L = [h_0, h_1, \ldots, h_{c-1}]$ a lista gerada, onde $h_k^1$ é o valor da soma dos pesos e $h_k^2$ é o valor do seu respectivo $ind$, $0 \leq k < c$. Cada processador $P_i$, $0 \leq i < p$, escreverá $h_j^2$ em $g_1[h_j^1]$ desde que $h_j^1 \neq h_{j-1}^1$, onde $i\lceil c/p \rceil \leq j < (i+1)\lceil c/p \rceil$. Para evitar possíveis conflitos de escrita no vetor $g_1$, $P_i$ testa no início se $h_{i\lceil c/p \rceil}^1 \neq h_{i\lceil c/p \rceil - 1}^1$.

O **Passo 1** por ser feito claramente em tempo $\mathcal{O}(q)$. No **Passo 2**, podemos utilizar um algoritmo de tempo $\mathcal{O}(q)$ [Col88]. No entanto, como se verá logo a seguir, outros algoritmos paralelos de ordenação de tempo $\mathcal{O}(q^2)$ também poderiam ser utilizados, pois isso não afetará a complexidade de tempo desta fase.

A análise da complexidade do **Passo 3** torna-se mais fácil considerando separadamente as iterações dos comandos **(3.1)** e **(3.2)**. A primeira iteração de **(3.1)** é executada em tempo constante, pois temos $p = 2^q$ processadores e uma lista com $2^q$ elementos: cada processador $P_j$, $0 \leq j < p$, ficaria com a incumbência de somar o valor de $w_i$ ao $j$-ésimo elemento da lista $L'$. No passo seguinte, este tempo será o dobro, pois a lista terá dobrado de tamanho; e assim por diante, até o último passo. Portanto, o tempo total gasto pelo comando **(3.1)** é $\sum_{k=0}^{\log_2 c - 1 - q} \mathcal{O}(2^k) = \mathcal{O}(2^{\log_2 c - q}) = \mathcal{O}(c/p)$.

Por outro lado, é sabido que, em uma máquina PRAM SIMD CREW com $p$ processadores, duas listas ordenadas de tamanho $k$ podem ser intercaladas em tempo $\mathcal{O}(k/p + \log k)$ (veja, por exemplo, [Akl89]). Além disso, é possível observar que o comando **(3.2)** é executado $\log_2 c - q + 1$ vezes e, para $p = 2^q$ processadores, é imediato concluir que o tempo gasto pelas iterações deste comando no **Passo 3** será $\mathcal{O}(2^{\log_2 c - q} + \log_2^2 c - q^2) = \mathcal{O}(\frac{c}{p} + \log^2 c)$.

Como o **Passo 4** também gasta tempo $\mathcal{O}(c/p)$, a complexidade da ***Fase 1*** será portanto $\mathcal{O}(\frac{c}{p} + \log^2 c)$.

***Fase 2:*** **Inserção de $n - 2\log_2 c$ pesos em $g_1$.** Uma vez que o vetor $g_1$ já está formado com todas as possíveis somas dos primeiros $\log_2 c$ pesos, a inserção de outros $n - 2\log_2 c$ pesos pode ser feita em paralelo com $\log_2(n - 2\log_2 c) \leq p \leq n - 2\log_2 c$ processadores através do Algoritmo $2a$ em tempo $\mathcal{O}(\frac{(n-2\log_2 c)(c-2w_{min})}{p} + \frac{c}{p} + n - \log c^2)$.

***Fase 3:*** **Geração de $g_2$ com as combinações dos últimos $\log_2 c$ pesos.** Esta fase é análoga à ***Fase 1***. No entanto, ao contrário do vetor $g_1$, $g_2$ será a própria lista ordenada com $c$ elementos, isto é, a lista $L$ da ***Fase 1***. Como já foi visto, esta lista $g_2$ pode ser gerada em tempo $\mathcal{O}(c/p)$.

***Fase 4:* Busca da solução ótima.** Utilizaremos novamente uma idéia apresentada no capítulo anterior. Inicialmente, é executada uma *fase de descarte*, que consiste em dividir ambas as listas em $p$ blocos de mesmo tamanho e selecionar os pares de blocos (um de cada lista) que podem conter a solução ótima. Com $p = 2^q$ processadores, a *fase de descarte* gasta tempo $\mathcal{O}(\log p) = \mathcal{O}(\log n)$, e seleciona um máximo de $2p$ pares de blocos.

A única diferença com o algoritmo do Capítulo IV é que, ao invés de duas listas ordenadas, temos agora um vetor $g_1$ e uma lista ordenada $g_2$. Aliás, vale a pena lembrar que $g_1$ e $g_2$ armazenam dados de natureza distinta: em $g_1$, há índices que variam entre 0 e $n$, enquanto que em $g_2$ estão armazenadas somatórias de pesos.

No entanto, ambas as estruturas têm o mesmo tamanho $c$ e, para a aplicação do algoritmo da *fase de descarte*, basta que cada processador $P_i$ encontre antes o maior e o menor elemento do $i$-ésimo bloco do vetor $g_1$. Isto gasta um tempo extra proporcional ao tamanho dos blocos, ou seja, $\mathcal{O}(c/p)$. A partir daí, as buscas binárias simultâneas poderão ser realizadas na lista $g_2$ sem problemas, bem como o resto da *fase de descarte*.

Na posterior e definitiva *fase de busca*, cada processador tem associado a si um ou dois pares de blocos. Como esses blocos têm tamanho $\lceil c/p \rceil$, o tempo desta fase será também $\mathcal{O}(c/p)$.

Portanto, o tempo total da ***Fase 4*** é $\mathcal{O}(\frac{c}{p} + \log n)$.

**Complexidade final.** As quatro fases totalizam tempo $\mathcal{O}(\frac{(n-2\log_2 c)(c-2w_{min})}{p} + \frac{c}{p} + n - \log c^2 + \log^2 c + \log n) = \mathcal{O}(\frac{(n-2\log_2 c)(c-2w_{min})}{p} + \frac{c}{p} + n - \log c^2 + \log^2 c)$, que é a complexidade final do Algoritmo 3. Além disso, ele gasta apenas espaço $\mathcal{O}(n+c)$.

## V.4 Quadro comparativo com os novos algoritmos

No quadro abaixo, temos as complexidades dos algoritmos que resolvem o Problema da Mochila numa PRAM SIMD: o algoritmo de Lin e Storer resolve o KP01 numa máquina EREW, enquanto os nossos resolvem o SSP exigindo leitura simultânea (modelo CREW). É fácil observar que os algoritmos 1, 2$a$ e 3 gastam tempo $o(nc/p)$.

| ALGORITMOS | TEMPO | ESPAÇO | PROCESSADORES |
|---|---|---|---|
| Lin e Storer | $\mathcal{O}(\frac{nc}{p})$ | $\mathcal{O}(nc)$ | $p \leq c+1$ |
| **Algoritmo** 1 | $\mathcal{O}(\frac{n}{p}(c-2w_{min}) + \frac{c+w_{max}-w_{min}}{p} + n + \log p)$ | $\mathcal{O}(n+c)$ | $p \leq w_{min}$ |
| **Algoritmo** 2$a$ | $\mathcal{O}(\frac{n}{p}(c-2w_{min}) + \frac{c}{p} + n)$ | $\mathcal{O}(n+c)$ | $\log n \leq p \leq n$ |
| Algoritmo 2$b$ | $\mathcal{O}(\frac{n}{p}(c-2w_{min}+\log n) + \frac{c}{p})$ | $\mathcal{O}(n+c)$ | $p \leq \log n$ |
| **Algoritmo** 3 | $\mathcal{O}(\frac{(n-2\log_2 c)(c-2w_{min})}{p} + \frac{c}{p} + n - \log c^2 + \log^2 c)$ | $\mathcal{O}(n+c)$ | $\log(n-2\log_2 c) \leq p \leq n - 2\log_2 c$ |

Apresentamos portanto os primeiros algoritmos paralelos adaptativos de toda a literatura que resolvem numa PRAM SIMD CREW de $p$ processadores uma variante do Problema da Mochila — no caso, o SSP de $n$ objetos e capacidade $c$ — em tempo $o(nc/p)$ e espaço $\mathcal{O}(n + c)$. Eles melhoram as complexidades de tempo e de espaço do algoritmo de Lin e Storer [LiS90, LiS91], que vinha sendo o mais eficiente até o momento. Estes resultados devem originar ainda uma nova publicação [SSY04].

# Capítulo VI

# Conclusões finais e perspectivas

Como contribuição original desta tese, apresentamos quatro novos algoritmos paralelos para uma máquina PRAM SIMD CREW que resolvem de modo eficiente uma importante variante do Problema da Mochila: o *Subset-Sum Problem* (SSP).

A importância do SSP e a relevância do modelo escolhido — a PRAM — já foram comentadas no capítulo inicial deste trabalho. Também foram apresentados os resultados de uma extensa pesquisa bibliográfica, que permitiu o levantamento dos melhores algoritmos seqüencias e paralelos desenvolvidos para a resolução exata do Problema da Mochila. Como pôde ser visto ao longo dos Capítulos II e III, todos esses algoritmos baseiam-se em apenas duas abordagens: no **paradigma das listas** e no **paradigma da programação dinâmica**.

No caso do paradigma das listas, a melhor resolução paralela do SSP até o momento devia-se a Ferreira e Robson [Fer96], que elaboraram um algoritmo adaptativo voltado para o hipercubo, mas que, em uma máquina PRAM SIMD EREW com $1 \leq p \leq 2^{n/4}$ processadores, gasta tempo $\mathcal{O}(n2^{n/2}/p)$ e espaço $\mathcal{O}(2^{n/4})$. Como foi comentado, a aceleração deste algoritmo paralelo perante o *algoritmo das duas listas* [HoS74] é $\mathcal{O}(p/n)$, ou seja, não é ótima.

No Capítulo IV, apresentamos uma paralelização ótima e adaptativa do *algoritmo das duas listas*: numa PRAM SIMD CREW de $p$ processadores, resolve o SSP de $n$ objetos em tempo $\mathcal{O}(2^{n/2}/p)$ e espaço $\mathcal{O}(2^{n/2})$, onde $1 \leq p < 2^{n/2}/n^2$. Este resultado, que é o primeiro de toda a literatura a alcançar aceleração ótima $\mathcal{O}(p)$ numa PRAM SIMD perante o *algoritmo das duas listas*, já foi anunciado numa conferência internacional [YSS01] e deverá ser publicado futuramente [SSY03]. É importante ressaltar que as pesquisas possibilitaram também um outro resultado já publicado [SSY02]: uma importante correção em dois artigos de um conceituado periódico internacional.

Considerando agora o paradigma da programação dinâmica, o melhor algoritmo para-

lelo adaptativo para resolver o KP01 (e, conseqüentemente, também o SSP) de $n$ objetos e capacidade $c$ numa PRAM SIMD EREW de $p$ processadores era o de Lin e Storer [LiS90, LiS91]: gastava tempo $\mathcal{O}(nc/p)$ e espaço $\mathcal{O}(nc)$. No Capítulo V, apresentamos três algoritmos paralelos adaptativos que resolvem o SSP numa PRAM SIMD CREW de $p$ processadores em menor tempo e exigem somente memória linear: gastam tempo $o(nc/p)$ e espaço $\mathcal{O}(n+c)$. Como estes algoritmos paralelos são os primeiros de toda a literatura a alcançar tal resultado numa PRAM SIMD, também deverão ser publicados proximamente [SSY04].

Um interessante campo de pesquisa que ainda se abre é a possibilidade de se adaptar a lógica desses novos algoritmos para outros modelos de Computação Paralela. Além disso, também tornam-se temas de pesquisa a aplicabilidade desses resultados, ou seja, sua implementação em computadores paralelos reais, com os necessários acertos nas complexidades e os correspondentes estudos comparativos.

Por fim, um comentário extra que não muda em nada as conclusões já apresentadas. Foi dito no primeiro capítulo que, de modo geral, a PRAM é considerada como um modelo teórico de Computação Paralela devido à inviabilidade prática de se construir máquinas com as suas características. O problema concreto deste modelo é possibilitar que cada processador tenha um acesso suficientemente rápido às posições da sua memória compartilhada. Como comenta Akl [Akl97], a questão crucial da PRAM está numa implementação eficiente da **unidade de acesso** à memória. No entanto, levando-se em conta tantos avanços tecnológicos já ocorridos na arquitetura de computadores, não parece absurdo supor que isso possa mudar no futuro. Uma hipótese possível é que os resultados advindos deste recente ramo da ciência que tem sido chamado de **nanotecnologia** — que propõe outras estruturas físicas de armazenamento e, conseqüentemente, novos modos de acesso aos dados — possam alterar tremendamente a fronteira da viabilidade prática. Se isso vier a acontecer, os algoritmos originais apresentados nesta tese não seriam somente uma contribuição para a **Teoria da Computação**, mas passariam a ter uma aplicabilidade direta.

# Referências Bibliográficas


[Akl84] S.G. Akl, "Optimal parallel algorithms for computing convex hulls and for sorting", *Computing* (33), pp. 1-11, 1984.

[Akl89] S.G. Akl, *The design and analysis of parallel algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Akl97] S.G. Akl, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, New Jersey, 1997.

[Bat68] Batcher, K.E., "Sorting Networks and their Applications", *Proceedings of AFIS 1968 SJCC*, AFIPS Press, Montvale, N.J., Vol. 32, p. 307-314, 1968.

[Bel57] R.E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957.

[Cha94] H.K.-C. Chang, J.J.-R. Chen and S.-J. Shyu, "A parallel algorithm for the knapsack problem using a generation and searching technique", *Parallel Computing*, pp. 233-243, 1994.

[Col88] R. Cole, "Parallel merge sort", *SIAM J. Comput.*, 17, pp. 770-785, 1988.

[Dan57] G.B. Dantzig, "Discrete variable extremum problems", *Operations Research*, 5, pp. 266-277, 1957.

[Fer91] A.G. Ferreira, "A parallel time/tradeoff $T.H = \mathcal{O}(2^{n/2})$ for the knapsack problem", *IEEE Trans. Comput.*, 40 (2), pp. 221-225, 1991.

[Fer96] A.G. Ferreira, "Parallel and communication algorithms for hypercube multiprocessors", *Handbook of Parallel and Distributed Computing*, A. Zomaya (Ed.), McGraw-Hill, New York, 1996.

[FeR96] A.G. Ferreira and J.M. Robson, "Fast and scalable parallel algorithms for knapsack-like problems", *Journal of Parallel and Distributed Computing*, 39, pp. 1-13, 1996.

[Fly66] M.J. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, 54, pp. 1901-1909, 1966.

[GaJ79] R. Garey and D.S. Johnson, *Computers and intractability: a guide to the theory of NP-Completeness*, Freeman, New York, 1979.

[Gre80] H. Greenberg, "An algorithm for a linear diophantine equation and a problem of Frobenius", *Numer. Math.*, 34, pp. 349-352, 1980.

[GoT97] A. Goldman and D. Trystram, "An efficient parallel algorithm for solving the knapsack problem on the hypercube", *Proceedings of 11th International Parallel Processing Symposium*, pp. 608-615, 1997.

[GoT03] A. Goldman and D. Trystram, "An efficient parallel algorithm for solving the knapsack problem on the hypercubes", a ser publicado no *Journal of Parallel and Distributed Computing.*

[GRK86] P.S. Gopalakrishnam, I.V. Ramakrishnam and L.N. Kanal, "Parallel approximate algorithms for the 0-1 knapsack problem", *Proceedings of International Conference on Parallel Processing*, pp. 444-451, 1986.

[HoS74] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem", *Journal of ACM*, pp. 277-292, 1974.

[JaJ92] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

[Kar84] E.D. Karnin, "A parallel algorithm for the knapsack problem", *IEEE Trans. Comput.*, C-33, pp. 404-408, 1984.

[KiL86] G.A.P. Kindervater and J.K. Lenstra, "An introduction to parallelism in combinatorial optimization", *Discrete Applied Mathematics*, 14, pp. 135-156, 1986.

[KGG94] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing – Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[LSS88] J. Lee, E. Shragowitz and S. Sahni, "A hypercube algorithm for the 0/1 knapsack problem", *Journal of Parallel and Distributed Computing*, 5, pp. 438-456, 1988.

[Leo01] C. Leopold, *Parallel and Distributed Computing - A Survey of Models, Paradigms and Approaches*, Wiley-Interscience, 2001.

[Lev89] E. Levin, "Grand challenges to computational science", *Communications of the ACM*, Vol. 32, n. 12, pp. 1456-1457, 1989.

[LiS90] J. Lin and J. Storer, "A new parallel algorithm for the knapsack problem and its implementation on a hypercube", *Proceedings of 3rd Symposium on the Frontiers of Massively Parallel Computation*, pp. 2-7, 1990.

[LiS91] J. Lin and J. Storer, "Processor efficient hypercube algorithm for the knapsack problem", *Journal of Parallel and Distributed Computing*, 3, pp. 332-337, 1991.

[LoC97] D.C. Lou and C.C. Chang, "A parallel two-list algorithm for the knapsack problem", *Parallel Computing*, pp. 1985-1996, 1997.

[May88] E.W. Mayr, "Parallel approximation algorithms", *Proceedings of International Conference on Fifth Generation Computer Systems*, pp. 542-551, 1988.

[MiA85] A. Mirzaian and E. Arjomandi, "Selection in X + Y and matrices with sorted rows and columns", *Inform. Process. Letters*, 20-1, pp. 13-17, 1985.

[NaS81] D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers", *IEEE Trans. Comput.*, C-30, pp. 101-107, 1981.

[NaS82] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network", *Journal of ACM*, pp. 642-667, 1982.

[Pis99] D. Pisinger, "Linear time algorithms for knapsack problems with bounded weights", *Journal of Algorithms*, 33, pp. 1-14, 1999.

[Qui94] M.J. Quinn, *Parallel Computing - Theory and Practice*, McGraw-Hill, 1994.

[SaS94] C.A.A. Sanches and S.W. Song, "SIMD algorithms for matrix multiplication on the hypercube", *Proceedings of 8th International Parallel Processing Symposium*, pp. 492-496, 1994.

[SSY02] C.A.A. Sanches, N.Y. Soma and H.H.Yanasse, "Comments on parallel algorithms for the knapsack problem", *Parallel Computing*, 28, pp. 1501-1505, 2002.

[SSY03] C.A.A. Sanches, N.Y. Soma and H.H. Yanasse, "An adaptive optimal parallel *two-list* algorithm for the knapsack problem", em elaboração.

[SSY04] C.A.A. Sanches, N.Y. Soma and H.H. Yanasse, "Algoritmos paralelos para o *Subset-Sum Problem* com tempo $o(nc/p)$ e espaço $\mathcal{O}(n+c)$", em elaboração.

[ScS81] R. Schroeppel and A. Shamir, "A $T = \mathcal{O}(2^{n/2}), S = \mathcal{O}(2^{n/4})$ algorithm for certain NP-complete problems", *SIAM J. Comput.*, 10 (3), pp. 456-464, 1981.

[SoT02] N.Y. Soma and P. Toth, "An exact algorithm for the subset sum problem", *European Journal of Operational Research*, 136, pp. 57-66, 2002.

[Sti95] D.R. Stinson, *Criptography: theory and practice*, CRC Press, Boston, 1995.

[Ten90] S. Teng, "Adaptive parallel algorithms for integral knapsack problems", *Journal of Parallel and Distributed Computing*, 8, pp. 400-406, 1990.

[Wil86] H.S. Wilf, *Algorithms and Complexity*, Prentice-Hall, 1986.

[YaS87] H.H. Yanasse and N.Y. Soma, "An exact pseudopolynomial algorithm for the value independent knapsack problem", *Proceedings of the XX SBPO*, 1, pp. 710-719, Salvador, 1987.

[YSS01] H.H. Yanasse, C.A.A. Sanches and N.Y. Soma, "An adaptive optimal parallel *two-list* algorithm for the knapsack problem", *The European Operational Research Conference*, Roterdam, 2001.