Implementações em Grades Computacionais de Algoritmos BSP/CGM para os Problemas da Mochila 0-1 e Mínimo Intervalar *

Edson N. Cáceres, Henrique Mongelli,
Christiane Nishibe e Hercules da C. Sandim
Universidade Federal do Mato Grosso do Sul,
Dept. de Computação e Estatística,
Campo Grande - MS, Brasil, 79070-900
{edson, mongelli, cnishibe, hcs}@dct.ufms.br

Resumo

Neste artigo, apresentamos os resultados experimentais das implementações dos algoritmos BSP/CGM para os problemas da mochila 0-1 e do mínimo intervalar, utilizando uma grade computacional. A grade computacional dos testes utiliza o middleware desenvolvido no Projeto Integrade. Estes algoritmos foram implementados usando a biblioteca BSPLib do Integrade. Para analisar o desempenho do middleware, os algoritmos também foram implementados num Beowulf (cluster) usando as bibliotecas BSP-Pub e LAM-MPI. Os resultados obtidos foram bastante promissores na validação do middleware, apresentando bons tempos de processamento.

1 Introdução

O Projeto InteGrade [14, 18] tem como o objetivo a construção de um *middleware* que permita a implantação de grades sobre recursos computacionais não dedicados, fazendo uso da capacidade ociosa normalmente disponível nos parques computacionais já instalados. O InteGrade é um projeto desenvolvido em conjunto por cinco instituições: Departamento de Ciência da Computação (IME-USP), Departamento de Informática (PUC-RIO), Departamento de Informática (UFMA), Instituto de Informática (UFG) e Departamento de Computação e Estatística (UFMS).

O InteGrade possui uma arquitetura orientada a objetos, onde cada módulo do sistema comunica-se com os demais a partir de chamadas remotas de métodos. O InteGrade utiliza CORBA [15] como sua infra-estrutura de objetos distribuídos, beneficiando-se de um substrato elegante e consolidado, o que se traduz na facilidade de implementação, uma vez que a comunicação entre os módulos do sistema é abstraída pelas chamadas aos métodos [25].

Desde sua concepção, o InteGrade tem como objetivo permitir o desenvolvimento de aplicações para resolver uma ampla gama de problemas paralelos. Vários sistemas de Computação em Grade restringem seu uso a problemas que podem ser decompostos em tarefas independentes, como *Bag-of-Tasks* [7] ou aplicações paramétricas. A fim de testar o *middleware* foram projetados algoritmos para vários problemas, entre eles o Problema da Mochila 0-1 e o Problema do Mínimo Intervalar que são problemas que têm pouca comunicação entre os processadores que tratamos neste artigo.

O Problema da Mochila 0-1 consiste em um conjunto $S = \{1, 2, \ldots, n\}$ de n itens distintos, cujo i-ésimo item possui um valor v_i em reais e um peso w_i em quilos, onde v_i e w_i são inteiros. Seja W um inteiro que representa a capacidade máxima da mochila que será utilizada para transportar os itens. A questão é: quais itens devem ser escolhidos a fim de encher a mochila com os itens mais valiosos sem exceder a capacidade máxima, isto é:

$$\max\{\sum_{i=1}^{n} v_i z_i : \sum_{i=1}^{n} w_i z_i \leq W, z_i \in \{0,1\}\}.$$

O Problema do Mínimo Intervalar consiste em pré-processar um vetor $A = (a_1, a_2, \ldots, a_n)$, de n números reais, de forma que consultas $MIN(i,j)$, para quaisquer $1 \leq i \leq j \leq n$, possam ser respondidas em tempo constante, onde $MIN(i,j) = \min\{a_i, \ldots, a_j\}$.

O modelo *Coarsed Grained Multicomputer* (CGM), proposto por Dehne et al. [8] consiste em um conjunto de p processadores, cada um com memória local de tamanho $O(n/p)$ e conectados por uma rede de interconexão, onde n é o tamanho do problema e $n/p \geq p$. Este modelo é uma simplicacão do modelo *Bulk Synchronous Parallel* (BSP)

*Apoiado por CNPq e FUNDECT.

proposto por Valiant [27], que também é um modelo dito realístico, pois define parâmetros para mapear as principais características de máquinas paralelas reais, ou seja, levando em consideração, dentre outras coisas, o tempo de comunicação entre os processadores. Denominamos de h−relação a quantidade h de dados trocados em uma comunicação.

Os algoritmos apresentados utilizam uma combinação desses dois modelos. Um algoritmo BSP/CGM possui rodadas de computação local alternadas com rodadas de comunicação entre os processadores, onde cada processador envia e recebe, em cada rodada, $O(n/p)$ dados no máximo. As rodadas de computação local e de comunicação entre os processadores são separadas por barreiras de sincronização.

O tempo de execução de um algoritmo BSP/CGM é a soma dos tempos gastos tanto com computação local quanto com comunicações entre os processadores. Nas rodadas de computação local, geralmente utilizamos o melhor algoritmo seqüencial para o processamento, além de estarmos interessados em minimizar o número de rodadas de computação.

Os problemas foram implementados utilizando a linguagem C em conjunto com as bibliotecas MPI, PUB/BSP e BSPLib/InteGrade.

Os algoritmos BSP/CGM foram executados no BIOPAD, um *cluster* composto por 12 nós: uma máquina AMD Athlon(tm) 1800+ 1GB de RAM; uma máquina Intel(R) Pentium(R) 4 CPU 1.70GHz 1GB de RAM; três máquinas Pentium IV 2.66GHz 512MB; uma máquina Pentium IV 2.8GHz 512MB; uma máquina Pentium IV 1.8GHz 480MB; quatro máquinas AMD Athlon(tm) 1.66GHz 480MB; uma máquina AMD Sempron(tm) 2600+ 480 MB. Os nós estão conectados por um *fast-Ethernet switch* de 1Gb. Cada nó executava o sistema operacional Fedora Core 3 com gcc-c++-3.4.4-2, LAM/MPI 7.1.1-7, PUB/BSP v8.0, InteGrade 0.2 RC4.

Para cada tamanho de entrada foi gerado um conjunto de dados aleatórios. Os tempos obtidos foram medidos em segundos e consideraram o tempo gasto com comunicação entre os precessadores e o tempo de computação local.

2 Problema da Mochila 0-1

Este problema clássico de otimização combinatorial possui uma enorme quantidade de aplicações [12, 17, 26]. Além disso, este é um problema de programação inteira com uma única restrição. Em princípio, qualquer problema de programação inteira pode ser transformado neste problema [12]. As dificuldades da solução do Problema da Mochila 0-1 são as dificuldades típicas da programação inteira. Bons algoritmos para o Problema da Mochila 0-1 são interessantes para a área de pesquisa em programação inteira.

O Problema da Mochila 0-1 pertence à classe dos problemas NP-*completos* [11]. Contudo, este problema pode ser resolvido seqüencialmente em tempo $O(nW)$. Este limite não é polinomial para o tamanho da entrada visto que $\lg W$ bits são necessários para codificar a entrada W. Esta solução é chamada de *pseudo-polinomial* [11].

Existem basicamente duas abordagens para encontrar a solução exata do Problema da Mochila 0-1: *programação dinâmica* (PD) e *branch-and-bound* (B&B). Quando os parâmetros v_i e w_i são gerados independentemente e temos um problema muito grande, a abordagem (B&B) é na média mais eficiente quando implementada em máquinas seqüencias [19]. Quando os parâmetros estão interligados, a abordagem PD comporta-se melhor do que B&B [5, 6]. O primeiro algoritmo para o Problema da Mochila baseado em programação dinâmica foi desenvolvido por Gilmore e Gomory [13].

O Algoritmo 1 soluciona seqüencialmente o Problema da Mochila 0-1 em tempo $O(nW)$. Algoritmos paralelos para este problema foram propostos por [4, 5, 9, 23, 26]. Apresentaremos um algoritmo BSP/CGM baseado na abordagem PD.

Denotaremos $f(r, c)$, com $1 \leq r \leq n$ e $0 \leq c \leq W$, os valores da solução ótima para o Problema da Mochila 0-1 com um conjunto de objetos $[1, r]$ e peso c. Conseqüentemente, $f(n, W)$ é o valor da solução ótima. A relação de recorrência é:

$$f(r, c) = \max\{f(r-1, c), f(r, c-w_r) + v_r\}$$

$\forall c$, com $0 \leq c \leq W$, onde $r = 1, 2, \ldots n$.

Algoritmo 1 MOCHILA 0-1

Entrada: (1) v_i e w_i, $1 \leq i \leq n$; (2) W; e (3) p.
Saída: $f(n, W)$

```
1: for c ← 1 to W do
2:    f(0, c) ← 0;
3: end for
4: for r ← 1 to n do
5:    for c ← 1 to W do
6:       if c < w_k then
7:          f(r, c) ← f(r − 1, c);
8:       else
9:          f(r, c) ← max{f(r, c − w_r) + v_k, f(r − 1, c)};
10:      end if
11:   end for
12: end for
```

Estamos interessados em descrever um algoritmo paralelo que possa ser implementado em máquinas paralelas disponíveis e obter tempos de execução compatíveis aos previstos no modelo BSP/CGM, independentemente do tipo

de interconexão de rede utilizado. Neste artigo, apresentamos um algoritmo BSP/CGM para o Problema da Mochila 0-1 que está baseado na idéia *wavefront* de [3]. Uma característica e vantagem da *wavefront* ou comunicação sistólica é que cada processador comunica-se com poucos processadores, o que o torna potencialmente mais conveniente para aplicações em grades computacionais. Nosso algoritmo gasta $O(\frac{nW}{p})$ de computação local e $O(p)$ rodadas de comunicação, onde p é o número de processadores.

2.1 O Algoritmo *Wavefront*

Nesta seção apresentamos um algoritmo BSP/CGM que gasta $O(p)$ rodadas de comunicação para computar a solução do problema da mochila 0-1 com n itens e peso máximo W. Utilizaremos p processadores, cada um com memória local $O(nW/p)$.

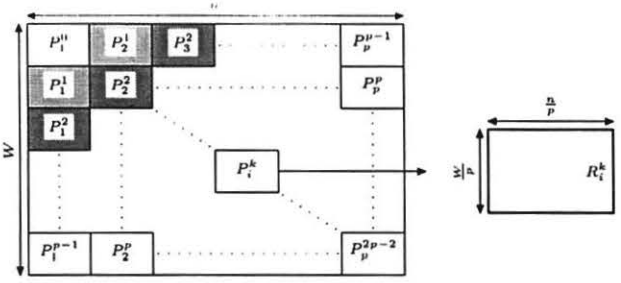

Figura 1. Um escalonamento de $O(p)$ rodadas de comunicação

Primeiro daremos a idéia principal para computar a matriz com a solução ótima f utilizando p processadores. Para cada conjunto $S = \{1, 2, \ldots, n\}$ de itens, o vetor w, onde $w[i]$ é o peso de cada item i, é distribuído para todos os processadores, o vetor v, onde $v[i]$ é o valor do item i, dividido em p partes, de tamanho $\frac{n}{p}$, e cada processador P_i, $1 \leq i \leq p$, recebe o i-ésimo parte de v ($v[(i-1)\frac{n}{p} + 1 \,..\, i\frac{n}{p}]$).

Esta idéia está ilustrada na Fig. 1. A notação P_i^k significa o trabalho do processador P_i na rodada k. Assim, inicialmente P_1 começa a computar na rodada 0. Então P_1 e P_2 podem trabalhar na rodada 1; P_1, P_2 e P_3 na rodada 2, e assim sucessivamente. Em outras palavras, após a computação da k-ésima parte da sub-matriz f_i (denotada por f_i^k), o processador P_i envia para o processador P_{i+1} os elementos da fronteira direita (coluna mais a direita) de f_i^k. Estes elementos são denotados por R_i^k. Utilizando R_i^k, o processador P_{i+1} pode computar a k-ésima parte da sub-matriz f_{i+1}. Após $p-1$ rodadas, o processador P_p recebe R_{p-1}^1 e computa a primeira parte da sub-matriz f_p. Na rodada $2p-2$, o processador P_p recebe R_{p-1}^p e computa a p-ésima parte da sub-matriz f_p e termina a computação.

É fácil verificar na tabela, que o processador P_p só inicia seu trabalho quando o processador P_1 termina sua computação, na rodada $p-1$. Portanto, temos um balanceamento de carga muito ruim.

Algoritmo 2 MOCHILA 0-1 PARALELO

Entrada: (1) O número p de processadores; (2) O número i do processador, onde $1 \leq i \leq p$; (3) A capacidade da mochila W e os subvetores v_i e w_i de tamanho $\frac{n}{p}$

Saída: $f(r,c) = \max\{f[r, c - w[r]] + v[r], f[r-1, c]\}$, onde $1 \leq c \leq W$ e $(j-1)\frac{n}{p} + 1 \leq r \leq j\frac{n}{p}$.

```
1:  for 1 ≤ k ≤ p do
2:    if i = 1 then
3:      for (k−1)W/p + 1 ≤ r ≤ kW/p and 1 ≤ c ≤ n/p do
4:        compute f(r,c);
5:      end for
6:      send(R_i^k, P_{i+1});
7:    end if
8:    if i ≠ 1 then
9:      receive(R_{i−1}^k, P_{i−1});
10:     for (k−1)W/p + 1 ≤ r ≤ kW/p and 1 ≤ c ≤ n/p do
11:       compute f(r,c);
12:     end for
13:     if i ≠ p then
14:       send(R_i^k, P_{i+1});
15:     end if
16:   end if
17: end for
```

Teorema 1 *O algoritmo 2 gasta $2p-2$ rodadas de comunicação com computação seqüencial de $O(\frac{Wn}{p})$ em cada processador.*

Prova. O processador P_1 envia R_1^k para o processador P_2 após computar o k-ésimo bloco $\frac{W}{p}$ de linhas da $\frac{Wn}{p}$ sub-matriz f_1. Após $p-1$ rodadas de comunicação, o processador P_1 termina seu trabalho. Similarmente, o processador P_2 acaba seu trabalho após p rodadas de comunicação. Assim, depois de $p-2+i$ rodadas de comunicação, o processador P_i finaliza seu trabalho. Visto que temos p processadores, após $2p-2$ rodadas de comunicação, todos os p processadores terão acabado seu trabalho.

Cada processador utiliza um algoritmo de programação dinâmica seqüencial para computar a solução ótima da sub-matriz f_i para o Problema da Mochila 0-1. Conseqüentemente este algoritmo gasta $O(\frac{Wn}{p})$. ∎

Teorema 2 *No final do algoritmo 2, $f(n, W)$ encontraremos a solução ótima para o Problema da Mochila 0-1 com n itens, valores v_i e pesos w_i, $1 \leq i \leq n$ e capacidade W.*

Prova. O Teorema 1 prova que após $2p - 2$ rodadas de comunicação, o processador P_p termina seu trabalho. Essencialmente, estamos computando um algoritmo de programação dinâmica seqüencial para o Problema da Mochila 0-1 e enviando as fronteiras para o processador da direita, a corretude do algoritmo aparece naturalmente com a corretude do algoritmo seqüencial. Portanto, após $2p - 2$ rodadas de comunicação, $f(n, W)$ armazenará a solução ótima para o Problema da Mochila 0-1 com n itens, valores v_i e pesos w_i, $1 \leq i \leq n$, capacidade W. ∎

2.2 Resultados Experimentais

A biblioteca MPI foi utilizada na Implementação 1 do Problema da Mochila 0-1; a biblioteca PUB/BSP [1] na Implementação 2; e a biblioteca BSPLib/InteGrade na Implementação 3.

A última linha das Tabelas 1, 2 e 3 mostram que o problema utiliza *swap* de memória quando $n = 8192$, $W = 32768$ e $p = 1$.

Na Implementação 1 é possível observar as vantagens do parelelismo, com exceção do caso em que $p = 2$, conforme aumentamos o número de processadores.

Tabela 1. Tempos Máximos obtidos com a Implementação 1

$W \times n$	1	2	4	8
2048 × 512	0.028	0.285	0.017	0.011
4096 × 1024	0.113	0.122	0.064	0.038
8192 × 2048	0.454	0.554	0.248	0.140
16384 × 4096	1.824	1.824	1.066	0.551
32768 × 8192	35.054	7.766	4.508	2.303

O número de sincronizações no PUB/BSP é maior que no MPI. Assim, na Implementação 2 há um crescimento no número de casos em que o tempo aumenta ao invés de diminuir.

Tabela 2. Tempo Máximos obtidos com a Implementação 2

$W \times n$	1	2	4	8
2048 × 512	0.014	0.022	0.015	0.018
4096 × 1024	0.056	0.082	0.049	0.133
8192 × 2048	0.226	0.325	0.184	0.228
16384 × 4096	0.877	1.351	0.708	0.506
32768 × 8192	42.870	5.556	3.081	1.773

Apesar de na Implementação 3, o número de barreiras de sincronização utilizadas no BSPLib/InteGrade ser ainda maior que o utilizado no PUB/BSP, os tempos são irregulares, havendo crescimentos e decrescimentos em relação à Implementação 2.

Tabela 3. Tempos Máximos obtidos com a Implementação 3

$W \times n$	1	2	4	8
2048 × 512	0.001	0.001	0.002	0.004
4096 × 1024	0.002	0.003	0.003	0.744
8192 × 2048	0.007	0.994	0.007	0.990
16384 × 4096	1.998	1.999	1.005	1.984
32768 × 8192	67.00	7.008	4.995	2.996

3 O Problema do Mínimo Intervalar

Dado um vetor de n números reais $A = (a_1, a_2, a_3, \ldots, a_n)$, definimos MIN($i$, j) = min(a_i, ..., a_j). O Problema do Mínimo Intervalar consiste em pré-processar o vetor de forma que as consultas MIN(i, j), para qualquer $1 \leq i \leq j \leq n$, possam ser respondidas em tempo constante.

3.1 Mínimo Prefixo e Mínimo Sufixo

Os problemas de mínimo prefixo e mínimo sufixo, são casos particulares de operações com prefixos e sufixos, onde consideramos a operação de mínimo [24].

Definição 1 *Considere um vetor $A = (a_1, a_2, \ldots, a_n)$ com n números reais. O mínimo prefixo do vetor A, é o vetor $P = (min(a_1), min(a_1, a_2), \ldots, min(a_1, a_2, \ldots, a_n))$. Analogamente, o mínimo sufixo do vetor A, é o vetor $S = (min(a_1, a_2, \ldots, a_{n-1}, a_n), min(a_2, \ldots, a_{n-1}, a_n), \ldots, min(a_{n-1}, a_n), min(a_n))$.*

3.2 O Problema do LCA (*Lowest Common Ancestor*)

Definição 2 *O ancestral comum mais baixo entre dois vértices u e v de uma árvore, é o vértice w, ancestral comum du e dv, que se encontra mais próximo da raiz. Denota-se por w = LCA(u, v).*

Definição 3 *O problema do LCA consiste no pré-processamento da árvore, tal que consultas LCA(u, v), para quaisquer vértices u e v da árvore, possam ser respondida em tempo constante.*

3.3 Algoritmos Seqüenciais

O algoritmo para o Problema do Mínimo Intervalar no modelo CGM, utiliza dois algoritmos seqüenciais, que são

executados utilizando os dados locais em cada processador. Estes algoritmos foram apresentados por Gabow *et al.* [10] e Alon e Schieber [2].

3.3.1 Algoritmo de Gabow *et al.*

Este algoritmo seqüencial utiliza a estrutura de dados denominada árvore cartesiana [28]. A árvore cartesiana de vetor $A = (a_1, a_2, a_3, \ldots, a_n)$, de n números reais distintos, é uma árvore binária cujos nós têm como rótulos os valores do vetor A. A raiz da árvore tem como rótulo a_m = MIN(a_1, a_2, ..., a_n) . Sua sub-árvore esquerda é uma árvore cartesiana para $A_{1,m-1} = (a_1, a_2, \ldots, a_{m-1})$, e sua sub-árvore direita é uma árvore cartesiana para $A_{m+1,n}$ = $(a_{m+1}, a_{m+2}, \ldots, a_n)$. A árvore para um vetor vazio é a árvore vazia.

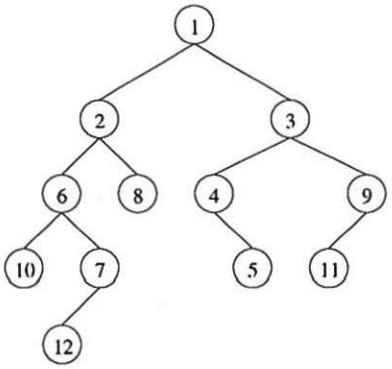

Figura 2. Árvore Cartesiana correspondente ao vetor (10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9).

Algoritmo 3 GABOW *et al.*

Entrada: (1) Um vetor de n números reais $A = (a_1, a_2, a_3, \ldots, a_n)$.

Saída: Uma estrutura de dados que responde as consultas MIN(i, j) em tempo constante.

1: Construir uma árvore cartesiana para A.
2: Aplicar um algoritmo seqüencial para o problema do LCA na árvore cartesiana A.

A partir da definição recursiva de árvore cartesiana, o valor de MIN(i, j) é o valor de LCA de a_i e a_j. Assim, cada consulta do Mínimo Intervalar pode ser respondida em tempo constante através de uma consulta LCA na árvore cartesiana. A construção da árvore cartesiana leva tempo linear. Existem vários algoritimos seqüenciais lineares para problemas de LCA. Logo o problema do Mínimo Intervalar é resolvido em tempo linear.

3.3.2 Algoritmo de Alon e Schieber [2]

O Algoritmo de Alon e Schieber tem complexidade $O(n \lg n)$. Apesar de sua complexidade não ser linear, este algoritmo é crucial na descrição do algoritmo CGM. Na descrição a seguir, considera-se que n é uma potência de 2, sem perda de generalidade.

Algoritmo 4 ALON E SCHIEBER

Entrada: (1) Um vetor de n números reais $A = (a_1, a_2, a_3, \ldots, a_n)$.

Saída: Uma estrutura de dados que responde as consultas MIN(i, j) em tempo constante.

1: Construir uma árvore binária completa T, denotada por Árvore-PS com n folhas.
2: Associar os elementos de A às folhas de T, da seguinte maneira:
3: **for** $v \in T$ **do**
4: Calcule os vetores P_v e S_v, onde P_v e S_v são os vetores que armazenam mínimo prefixo e máximo sufixo, respectivamente, dos elementos das folhas da sub-árvore com raiz em v.
5: **end for**

O procedimento para obtenção de P_v está descrito no procedimento ConstróiP', como visto no Algoritmo 5.

Algoritmo 5 PROCEDIMENTO CONSTRÓIP'

```
1:  P'[0] ← b_i
2:  apontador ← i
3:  inordem ← i + 1
4:  for k of 1 until lg p do
5:    P'[k] ← P'[k - 1]
6:    if ⌊inordem / 2^k⌋ mod 2 = 0 then
7:      for l of 1 until 2^(k-1) do
8:        apontador ← apontador - 1
9:        if P'[k] = B[apontador] then
10:         P'|k| ← B|apontador|
11:       end if
12:     end for
13:   end if
14: end for
```

O procedimento para obtenção de S_v é análogo ao procedimento ConstróiP' e denotado por ConstróiS'.

Para determinar MIN(i, j), encontramos w = LCA(a_i, a_j) em T. Sejam v e u os filhos esquerdo e direito de w, respectivamente. Então, MIN(i, j) é o mínimo entre o valor de S_v na posição correspondente a a_i e o valor de P_u na posição correspondente a a_j. O tempo constante para se executar uma consulta LCA em T vem do fato de esta árvore ser binária completa.

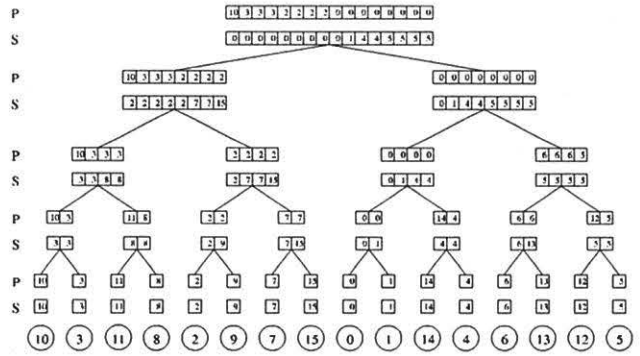

Figura 3. Árvore-PS gerada pelo algoritmo de Alon e Schieber para o vetor (10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9).

3.4 Algoritmo CGM

A idéia do algoritmo baseia-se nas possibilidades de como as consultas MIN(i, j) podem ser executadas. Cada processador armazena $\frac{n}{p}$ posições contíguas do vetor A. Denotamos por A_i, como sendo o subvetor armazenado pelo processador i.

Dependendo da localização de a_i e a_j nos processadores, temos os seguintes casos:

1. a_i **e** a_j **estão em um mesmo processador**: Nesse caso precisamos de uma estrutura eficiente para responder a essa consulta em tempo constante. Esta estrutura é obtida em cada processador através do Algoritmo de Gabow *et al.* [10].

2. a_i **e** a_j **estão em processadores diferentes**, $\bar{i}$ e $\bar{j}$, respectivamente: Nesse casos, teremos dois subcasos:

   - $i = j - 1$: a_i e a_j estão em processadores vizinhos. Nesse caso, MIN(i, j) consiste em se calcular o mínimo de a_i até o final do subvetor $A_{\bar{i}}$, e calcular o mínimo do início do subvetor $A_{\bar{j}}$ até a_j. Para se calcular o mínimo dos mínimos, precisamos de uma rodada de comunicação.
   - $i < j - 1$: MIN(i, j) corresponde ao mínimo entre os mínimos do subvetor $A_{\bar{i}}|\mathrm{i} \ldots (\bar{i}+1).\frac{n}{p}|$, o mínimo de $A_{\bar{j}}[\bar{j}.\frac{n}{p} \ldots \mathrm{j}]$ e os mínimos de A_{i+1}, ..., A_{j-1}. Os mínimos de A_{i+1}, ..., A_{j-1} são facilmente calculados com o uso da árvore cartesiana [28]. Para se calcular o mínimo dos mínimos, precisamos de uma estrutura que responda a consulta em tempo constante. Por isso, podemos utilizar o algoritmo de Alon e Schieber [2], já que o vetor de mínimos possui apenas p valores. A dificuldade nesse caso, é que não se pode construir a árvore binária completa T explicitamente em um processador (ou em todos), pois isso gastaria uma memória de tamanho $O(p \log p)$, que infringe as limitações do modelo CGM. Para contornarmos este problema, constrói-se P' e S' de $\log p + 1$ posições cada um, que armazenam algumas informações de T em cada processador.

Algoritmo 6 ALGORITMO CGM

Entrada: (1) Um vetor de n números reais $A = (a_1, a_2, a_3, \ldots, a_n)$.

Saída: Uma estrutura de dados que responde as consultas MIN(i, j) em tempo constante.

1: Cada processador i executa o algoritmo de Gabow *et al.* [10] seqüencialmente.

2: Cada processador constrói um vetor $B = (b(i))$ de tamanho p, que conterá o mínimo dos dados armazenados em cada processador:

   - Cada processador i calcula b_i = MIN A_i = MIN($a_{i.\frac{n}{p}+1}, \ldots, a_{(i+1).\frac{n}{p}}$).
   - Cada processador i envia b_i para os outros processadores.
   - Cada processador i coloca o valor recebido do processador k, $k \in \{0, 1, ..p-1\} \backslash \{i\}$, em b_k.

3: Cada processador i executa os procedimentos ConstróiP' e ConstróiS', que são análogos.

A complexidade do Algoritmo CGM é descrita abaixo:

- O passo 1 é executado em tempo $O(\frac{n}{p})$ e não realiza comunicação.
- O passo 2 roda em tempo seqüencial de $O(\frac{n}{p})$ e efetua uma rodada de comunicação.
- O passo 3 roda em tempo seqüencial de $O(\frac{n}{p})$ e não realiza comunicação.

3.5 Resultados Experimentais

A Implementação 1 do Mínimo Intervalar utilizou a biblioteca MPI; a Implementação 2 a biblioteca PUB/BSP; e a biblioteca BSPLib/InteGrade foi utilizada na Implementação 3.

Os tempos obtidos com a Implementação 2 (Tabela 5) assemelharam-se bastante aos tempos da Implementação 1 (Tabela 4), obtendo valores melhores para entradas maiores.

Os tempos obtidos com a Implementação 3 (Tabela 6) não foram melhores. O desempenho irregular da

Tabela 4. Tempos Máximos obtidos com a Implementação 1

n	1	2	4	8
32768	0.107	0.053	0.028	0.015
65536	0.215	0.109	0.056	0.029
131072	0.441	0.216	0.110	0.057
262144	0.866	0.426	0.223	0.115
524288	1.720	0.859	0.442	0.229
1048576	3.408	1.719	0.871	0.457
2097152	6.867	3.447	1.761	0.917

Tabela 5. Tempos Máximos obtidos com a Implementação 2

n	1	2	4	8
32768	0.060	0.050	0.065	0.093
65536	0.117	0.100	0.129	0.107
131072	0.235	0.195	0.139	0.130
262144	0.466	0.390	0.237	0.139
524288	0.934	0.780	0.435	0.238
1048576	1.866	1.607	0.823	0.471
2097152	3.725	3.175	1.605	0.865

Implementação 3 deve-se ao grande número de módulos existentes no InteGrade. Para a execução de um programa no InteGrade, são necessários que alguns módulos sejam executados concorrentemente, a fim de se realizar o monitoramento da execução, o escalonamento dos processos, entre outras atividades, o que pode deixar a execução do programa mais lenta.

Tabela 6. Tempos Máximos obtidos com a Implementação 3

n	1	2	4	8
32768	0.054	0.056	0.056	0.115
65536	0.101	0.110	0.057	0.383
131072	0.198	0.206	0.079	0.385
262144	0.390	0.499	0.259	0.300
524288	0.798	1.049	0.429	0.600

Apesar do *overhead* inerente à utilização do InteGrade por causa da quantidade de módulos para o gerenciamento da grade, seu desempenho, apesar de irregular não foi muito superior ao obtido com a Implementação 2, chegando a ser inferior em algumas instâncias.

O Mínimo Intervalar, por ter apenas uma rodada de comunicação, pôde usufruir das características da grade e servir como uma aplicação teste.

Na Implementação 3 não foi possível executar entradas, cujos valores eram $n = 1048576$ e $n = 2097152$, por uma limitação das máquinas.

4 Conclusões e Trabalhos Futuros

Os problemas apresentados neste artigo são problemas que não se enquadram na categoria *Bag-of-Tasks* e seus algoritmos BSP/CGM são conhecidos e foram implementados usando várias bibliotecas de troca de mensagens. Os experimentos foram realizados utilizando clusters e o *middleware* InteGrade.

Pôde-se observar que as implementações mostraram bons resultados na grade em comparação aos *clusters*, mesmo com os *overhead* dos módulos do InteGrade.

Outras implementações de algoritmos BSP/CGM estão sendo desenvolvidas e testadas no *middleware*. Entre estas estão versões de algoritmos para o Problema de Ordenação [20] e algoritmos FPT (*Fixed Parameter Tractability*) para o Problema da k-Cobertura de Vétices [16, 21].

Os resultados obtidos em cada problema são descritos nas respectivas seções de resultados experimentais. De forma geral, o desempenho usando o InteGrade foi bom. Resultados melhores são esperados para aplicações que utilizem a característica de *Bag-of-Tasks* das grades. As grades, em particular o InteGrade, não estão preocupados somente com o desempenho mas também com a utilização de recursos ociosos e disponibilização de grande volume de recursos (processamento e memória).

Referências

[1] Pub-library. Disponível em http://wwwcs.uni-paderborn.de/~bsp/, 2006. Último acesso em 31/03/2006.

[2] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Tel Aviv, Israel 69978, 1987. Tel Aviv University.

[3] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Proceedings ICCSA 2003*, volume 2668 of *Lecture Notes in Computer Science*, pages 249–258. Springer, 2003.

[4] R. Andonov, F. Raimbault, and P. Quinton. Dynamic programming parallel implementations for knapsack problem. Technical Report RI 740, IRISA, 1993.

[5] G. Chen, M. Chern, and J. Jang. Pipeline architectures for dynamic programming algorithms. *Parallel Computing*, 13:111–117, 1990.

[6] C. Chung, , M. S. Hung, and W. O. Rom. A hard knapsack problem. *Naval Research Logistics*, 35:85–98, 1988.

[7] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. *Lecture Notes in Computer Science*, 2790:169 – 180, 2003.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of the ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[9] A. Ferreira and J. M. Robson. Fast and scalable parallel algorithms for knapsack-like problems. *J. Parallel Distrib. Comput.*, 39:1–13, 1996.

[10] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM Press. *apud* [22].

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979. *apud* [16].

[12] R. Garfinkel and G. Nemhauser. *Integer Programming*. John Wiley and Sons, 1972.

[13] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

[14] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[15] O. M. Group. *CORBA v3.0 Specification*. Needham, MA, July 2002. OMG Document 02-06-33.

[16] E. J. Hanashiro. O problema da k-Cobertura por Vértices: Uma Implementação FTP no modelo BSP/CGM. Master's thesis, 2004.

[17] T. C. Hu. *Combinatorial Algorithms*. Addison Wesley, 1982.

[18] InteGrade. http://gsd.ime.usp.br/integrade, 2004.

[19] S. Martello and P. Toth. *Kanpsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[20] H. Mongelli and L. Gonda. Uma implementação de algoritmos bsp/cgm de ordenação. *VI Workshop em Sistemas Computacionais de Alto Desempenho*, 1(89–96), 2005. Anais do VI Workshop em Sistemas Computacionais de Alto Desempenho - WSCAD 2005.

[21] H. Mongelli, E. J. Hanashiro, and S. Song. Efficient implementation of the bsp/cgm parallel vertex cover fpt algorithm. *Third International Workshop on Experimental and Efficient Algorithms - WEA 2004*, 3059(253–268), 2004. Lecture Notes in Computer Science.

[22] H. Mongelli and S. Song. Parallel range minima on coarse grained multicomputers. *International Journal of Foundations of Computer Science*, 10(4):375–389, 1999.

[23] D. Morales, J. Roda, F. Almeida, C. Rodrigues, and F. Garcia. Integral knapsack problems: Parallel algorithms and theirs implementations on distributed systems. In *Proc. of the ACM-ICS 95*, pages 218–226, 1995.

[24] J. H. Reif. *Synthesis of Parallel Algorithms*. editor (Morgan Kaufmann Publishers), 1993. *apud* [22].

[25] R. A. Sevenich. Parallel processing using pvm. *Linux J.*, 1998(45es):2, 1998. *apud* [22].

[26] S. Teng. Adaptative parallel algorithm for integral knapsack problems. *J.of Parallel and Distributed Computing*, 14:1045–1074, 1990.

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

[28] J. Vuillemin. A unified look at data structures. pages 229–239, 1980. *apud* [22].