Uma Implementação de Algoritmos BSP/CGM de Ordenação

Luciano Gonda
Universidade Católica Dom Bosco
Departamento de Engenharia de Computação
Campo Grande - MS
gonda@ec.ucdb.br

Henrique Mongelli
Universidade Federal de Mato Grosso do Sul
Departamento de Computação e Estatística
Campo Grande - MS
mongelli@dct.ufms.br

Resumo

Neste trabalho descrevemos três algoritmos paralelos de ordenação: Ordenação_Bitônica, Ordenação_CD *e* Ordenação_Divisão, *desenvolvidos no modelo CGM e suas respectivas implementações. No CGM, cada processador possui memória local de tamanho* $O(\frac{n}{p})$ *e em cada rodada de comunicação cada processador pode enviar ou receber* $O(\frac{n}{p})$ *dados, onde* n *é o tamanho da entrada e* p *é o número de processadores utilizados. O algoritmo* Ordenação_Bitônica *utiliza* $O(\log p)$ *rodadas de comunicação e tempo de computação local* $O(\frac{n \log n}{p})$ *e sua implementação apresenta um bom desempenho em relação ao algoritmo seqüencial. Além disso, não foi encontrada nenhuma outra implementação deste algoritmo no modelo CGM e este algoritmo pode ser utilizado em ordenação externa. Os algoritmos* Ordenação_CD *e* Ordenação_Divisão *utilizam* $O(1)$ *rodadas de comunicação e tempo de computação local* $O(\frac{n \log n}{p})$. *Entretanto, o algoritmo* Ordenação_CD *apresenta um desempenho um pouco melhor. A implementação do deste algoritmo apresenta resultados muito bons em relação ao tempo de execução, mostrando que este algoritmo é eficiente se executado para tamanhos de entradas grandes.*

1. Introdução

A ordenação é uma das operações mais executadas na área de computação, pois a manipulação de dados ordenados, na maioria das aplicações, é mais fácil que a manipulação de dados em uma ordem arbitrária.

O problema de ordenação pode ser definido como a reorganização de uma coleção de elementos em ordem crescente ou decrescente. Knuth [8], define formalmente ordenação da seguinte forma:

Definição 1.1 *Um conjunto de elementos satisfaz uma ordem linear* $<$ *se e somente se:*

1. *quaisquer dois elementos* a *e* b *temos* $a < b$, $a = b$ *ou* $b < a$;
2. *quaisquer três elementos* a, b *e* c, *se* $a < b$ *e* $b < c$, *então* $a < c$.

A ordenação é muito importante para outros problemas, especialmente na busca de informações. Em paralelo, a ordenação é utilizada como sub-rotina para resolver diversos problemas em grafos [5].

O modelo *Coarsed Grained Multicomputer* (CGM), proposto por Dehne *et al.* [4] consiste em um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{n}{p})$ e conectados por uma rede de interconexão, onde n é o tamanho do problema e $\frac{n}{p} \geq p$. Este modelo é uma simplificação do modelo *Bulk Synchronous Parallel* (BSP) proposto por Valiant [9], 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.

Um algoritmo CGM possui rodadas de computação local alternadas com rodadas de comunicação entre os processadores, onde cada processador envia e recebe, em cada rodada, $O(\frac{n}{p})$ 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 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.

A ordenação é um importante problema que recorrentemente é explorado. Neste trabalho, apresentaremos três algoritmos paralelos de ordenação e suas respectivas implementações.

O algoritmo *Ordenação_Bitônica*, descrito na Seção 2, é baseado na idéia do algoritmo seqüencial apresen-

tado em Cáceres *et al.* [3]. Este algoritmo apresenta um bom desempenho em relação ao seqüencial, além disso pode ser aplicado na ordenação externa e suas rodadas de comunicação são sempre entre pares de processadores, não havendo comunicações coletivas. Este algoritmo foi descrito em diversos outros modelos, por exemplo, no modelo LogP [1], porém não existe nenhuma implementação no CGM.

O algoritmo *Ordenação_CD*, apresentado na Seção 3 utiliza a idéia de separar os elementos dos processadores em intervalos definidos por divisores globais. Este algoritmo possui um ótimo desempenho, principalmente, em decorrência de possuir um número constante de rodadas de comunicação.

Na Seção 4, o algoritmo *Ordenação_Divisão* será descrito. Este algoritmo divide os elementos pelos processadores utilizando p-quartis. Este algoritmo também apresenta um bom desempenho, pois o número de rodadas de comunicação é constante.

É importante observar que todos os algoritmos foram implementados na linguagem C/C++ juntamente com uma biblioteca de troca de mensagens MPI. Os experimentos foram executados em um Beowulf de 64 processadores Pentium III 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Os tempos obtidos para ordenação foram medidos em segundos, não incluindo o tempo de distribuição dos dados. Foram realizados 30 experimentos e os maiores tempos foram utilizados para a construção dos gráficos. Os arquivos de entrada foram os mesmos para todos os algoritmos e foram gerados aleatoriamente, onde para cada entrada foram utilizados 1, 2, 4, 8, 16 e 32 processadores.

Por fim, na Seção 5, realizamos as considerações finais sobre os algoritmos apresentados.

Os resultados completos e as demosntrações dos teoremas podem ser encontrados em [7].

2. Ordenação Bitônica

Nesta seção descreveremos um algoritmo de ordenação denominado de *Ordenação_Bitônica* que utiliza a idéia de unir pares de subseqüências , alternada com ordenações locais. A entrada do algoritmo é uma seqüência de números de tamanho n e utiliza p processadores, onde n é potência de dois e $\frac{n}{p} \geq p$.

Definição 2.1 *Uma seqüência de números* $(a_1, a_2, ..., a_n)$ *é dita bitônica, se existe um inteiro* $1 \leq j \leq n$, *tal que* $a_1 \leq a_2 \leq \ldots \leq a_j \geq a_{j+1} \geq \ldots \geq a_n$ *[3, 10].*

Uma seqüência de números também é denominada bitônica se pudermos deslocá-la ciclicamente, tal que a seqüência resultante S seja bitônica, ou seja, existe um número inteiro $1 \leq k \leq n$, tal que $S_{(1)} \leq S_{(2)} \leq \ldots \leq S_{(k)} \geq S_{(k+1)} \geq \ldots \geq S_{(n)}$.

Exemplo 2.1 *A seqüência (1, 3, 5, 6, 7, 4, 2) é bitônica, com* $k = 5$. *A seqüência (9, 6, 3, 2, 5, 7, 10) é bitônica, pois a seqüência (2, 5, 7, 10, 9, 6, 3) é bitônica com* $k = 4$.

Teorema 2.1 *Seja* $S = (a_1, a_2, \ldots, a_{2n})$ *uma seqüência bitônica. Podemos obter duas seqüências bitônicas aplicando a operação de* ***divisão bitônica*** *da seguinte forma:*

$S_{min} = (min\{a_1, a_{n+1}\}, \ldots, min\{a_n, a_{2n}\})$

e

$S_{max} = (max\{a_1, a_{n+1}\}, \ldots, max\{a_n, a_{2n}\})$.

Além disso, temos que $max(S_{min}) \leq min(S_{max})$, *ou seja, todos os elementos de* S_{min} *são menores ou iguais aos elementos de* S_{max} *[7].*

Exemplo 2.2 *Dada S = (2, 3, 6, 7, 8, 5, 4, 1), obtemos as seguintes seqüências bitônicas:* S_{min} *= (2, 3, 4, 1) e* S_{max} *= (8, 5, 6, 7).*

2.1. Descrição do Algoritmo

No algoritmo *Ordenação_Bitônica*, assumimos que o número de elementos n da seqüência a ser ordenada e o número de processadores p são potências de dois. A cada rodada do algoritmo, cada processador possui exatamente $\frac{n}{p}$ elementos da seqüência global.

A idéia do algoritmo que descreveremos é baseada na operação de divisões bitônicas sucessivas e ordenações locais, até que toda a seqüência esteja ordenada. É importante observar que, em paralelo, a operação de divisão bitônica é sempre executada entre pares de processadores, de forma que, após cada operação, os elementos da seqüência bitônica S_{min} ficarão armazenadas em um dos processadores, enquanto os elementos da seqüência bitônica S_{max} ficarão armazenados em outro processador.

Algoritmo: Ordenação_Bitônica

Entrada: uma seqüência de n elementos, distribuída entre os p processadores, rotulados de 0 a $p-1$ com $\frac{n}{p}$ dados em cada processador. Durante o algoritmo os processadores são re-rotulados localmente da forma $P_{g,i}$, onde g é o rótulo do grupo a que pertencem e i é o rótulo do processador dentro do grupo.

Saída: A seqüência ordenada de elementos distribuída entre os p processadores.

1. Cada processador ordena seus dados localmente. Os processadores de identificador par ordenam em ordem crescente e os processadores de identificador ímpar ordenam os dados em ordem decrescente.
2. **para** i **de** 1 **ate** $\log p - 1$ **faça**

2.1. $k = 2^{i-1}$

2.2. Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacentes cada. Os grupos são rotulados de 0 a $g - 1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t - 1$.

2.3. Execute em paralelo, em cada um dos g grupos, a operação de divisão bitônica entre $P_{g,j}$ e $P_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, S_{min} ficará armazenada no processador de menor índice global e S_{max} ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, S_{min} ficará armazenada no processador de maior índice global e S_{max} ficará armazenada no processador de menor índice global.

2.4. Cada processador ordena seus dados localmente,onde os processadores pertencentes aos grupos de rótulo par, ordenam os dados em ordem crescente e os processadores que pertencem aos grupos de rótulo ímpar, ordenam os dados em ordem decrescente.

3. **para** i **de** 1 **até** $\log p$ **faça**

3.1. $k = \frac{p}{2^i}$

3.2. Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacentes cada. Os grupos são rotulados de 0 a $g - 1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t - 1$.

3.3. Execute em paralelo, em cada um dos g grupos, a operação de divisão bitônica entre $P_{g,j}$ e $P_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, S_{min} ficará armazenada no processador de menor índice global e S_{max} ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, S_{min} ficará armazenada no processador de maior índice global e S_{max} ficará armazenada no processador de menor índice global.

4. Cada processador ordena localmente seus elementos em ordem crescente.

fim algoritmo

Teorema 2.2 *O algoritmo* Ordenação_Bitônica *ordena corretamente n inteiros armazenados em uma máquina CGM de p processadores, com $\frac{n}{p}$ inteiros por processador, utilizando-se $O(\log p)$ rodadas de comunicação ($\log p$ $\frac{n}{p}$-relações) e tempo de computação local $O(\frac{n \log n}{p})$ [7].*

É importante observar, que apesar de utilizar a operação de divisão bitônica, a entrada deste algoritmo não precisa ser necessariamente uma seqüência bitônica, pois o mesmo transforma a seqüência de entrada em uma seqüência bitônica, utilizando $\log p - 1$ passos, onde em cada passo i são executadas i rodadas de comunicação.

Para ilustrar o funcionamento do algoritmo *Ordenação_Bitônica*, considere a seqüência de elementos (7, 3, 9, 14, 18, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11), distribuída por 4 processadores. A Figura 1 mostra a execução do algoritmo para ordenar esta seqüência.

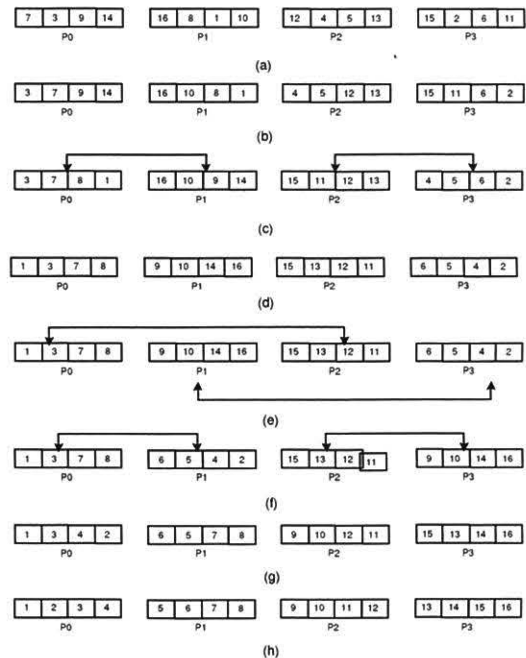

Figura 1. Exemplo dos passos do algoritmo *Ordenação_Bitônica*, com $p = 4$ e $n = 16$.

No exemplo da Figura 1, são necessários 2 passos para ordenação dos elementos. O primeiro passo transforma a seqüência de entrada em uma seqüência bitônica, utilizando uma rodada de comunicação. No segundo passo, são utilizadas duas rodadas de comunicação para ordenar a seqüência obtida no passo 1.

Inicialmente, os elementos são distribuídos pelos processadores (Figura 1*(a)*). Após a distribuição, os elementos são ordenados localmente. Obtemos duas seqüências bitônicas, uma formada pelos elementos em P_0 e P_1 e outra formada pelos elementos em P_2 e P_3, como mostra a Figura 1*(b)*.

Em seguida, é executada uma rodada de comunicação, na qual duas operações de divisão bitônica são executadas em paralelo, uma delas entre os processadores P_0 e P_1 (P_{0+1}) e outra entre os processadores P_2 e P_3 (P_{2+1}), pois $k = 1$(Figura 1*(c)*). Na operação de divisão bitônica entre os processadores P_0 e P_1, os elementos de S_{min} são armazenados em P_0 e os de S_{max} em P_1. Na divisão bitônica entre P_2 e P_3, os maiores elementos ficam em P_2 e os menores ficam em P_3. Isto é realizado para que tenhamos novamente uma seqüência bitônica, porém envolvendo os elementos dos quatro processadores. Finalizada a rodada de

comunicação, cada processador executa uma ordenação local. A configuração após esse passo é mostrada na Figura *1(d)*, onde temos uma seqüência bitônica formada entre os processadores P_0, P_1, P_2 e P_3.

Neste instante, a seqüência de entrada foi transformada em uma seqüência bitônica e podemos aplicar novamente operações de divisão bitônica. Como $k = 2$, serão executadas operações de divisão bitônica entre P_0 e P_2, e entre P_1 e P_3 (Figura *1(e)*). Após estas operações, os elementos ficam distribuídos como mostra a Figura *1(f)* e k passa a ser igual a 1. A seguir, são executadas operações de divisão bitônica entre os processadores P_0 e P_1 e entre P_2 e P_3, como mostra a Figura *1(f)*, obtendo-se os dados mostrados na Figura *1(g)*. Por fim, cada processador executa uma ordenação local. Após a ordenação, temos todos os elementos ordenados, distribuídos pelos processadores, como ilustrado na Figura *1(h)*.

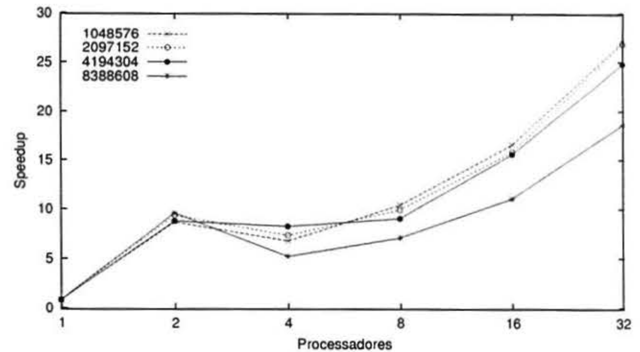

Figura 3. *Speedup* do algoritmo *Ordenação_Bitônica*, $1048576 \leq n \leq 8388608$.

2.2. Implementação e Resultados

Na Figuras 2 e 3 apresentamos, respectivamente, os tempos de execução e o *speedup* do algoritmo *Ordenação_Bitônica* para $1048576 \leq n \leq 8388608$, onde n é uma potência de 2.

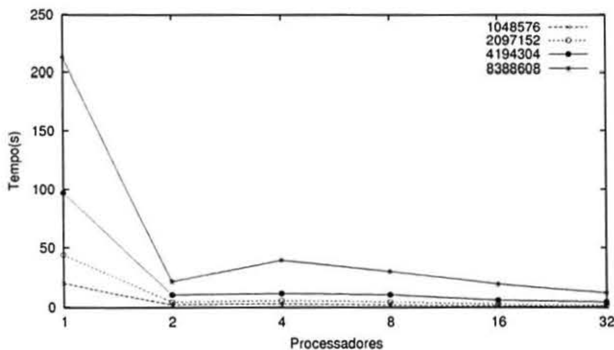

Figura 2. Tempo de execução do algoritmo *Ordenação_Bitônica*, $1048576 \leq n \leq 8388608$.

Observamos que mesmo aumentando o tamanho da entrada, continuamos tendo um desempenho ruim com poucos processadores, devido ao custo de comunicação entre os mesmos e também ao custo de computação local que é maior quando temos poucos processadores. Porém, à medida que aumentamos o número de processadores, o desempenho do algoritmo melhora e o *speedup* também aumenta. Além disso, o algoritmo paralelo apresenta um ganho significativo em relação ao seqüencial.

Para alguns casos o *speedup* é superlinear, pois o algoritmo de ordenação bitônica seqüencial é lento e nas computações locais do algoritmo paralelo usamos o *quicksort*.

3. Algoritmo Ordenação_CD

O algoritmo apresentado nesta seção, que denominamos de *Ordenação_CD*, foi proposto por Chan e Dehne [2] e baseia-se no método de ordenação por amostragem, combinado com um algoritmo de ordenação básico para computação local.

Apresentamos na Seção 3.1, o algoritmo para ordenação de inteiros, com $\frac{n}{p} \geq p^2$, onde n é o tamanho da entrada e p é o número de processadores utilizados. Na Seção 3.2, descrevemos os testes realizados, assim como os resultados obtidos na implementação do algoritmo.

3.1. Descrição do Algoritmo

Nesta seção, descreveremos o algoritmo *Ordenação_CD* de forma mais detalhada. Este algoritmo utiliza a idéia da ordenação por amostragem, em conjunto com um algoritmo de ordenação local.

Algoritmo: Ordenação_CD
Entrada: n elementos divididos em p processadores, com $\frac{n}{p}$ elementos em cada processador.
Saída: os elementos ordenados, divididos pelos processadores.

1. Cada processador P_i ordena localmente seus $\frac{n}{p}$ dados, com $1 \leq i \leq p$, usando um algoritmo de tempo $O(n \log n)$.

2. Cada processador P_i seleciona uma amostra local S_i de p elementos com índice $j\frac{n}{p^2}$, onde $1 \leq i \leq p$ e $0 \leq j \leq p-1$.
3. Cada processador P_i envia o S_i para processador P_1, onde $1 \leq i \leq p$.
4. Processador P_1 ordena os elementos recebidos no passo anterior, utilizando um algoritmo de tempo $O(n \log n)$ e armazena no vetor S.
5. Processador P_1 seleciona uma amostra G de p elementos contidos em S, com identificador jp, onde $0 \leq j \leq p$.
6. Processador P_1 envia G para todos os processadores.
7. Cada processador P_i separa seu conjunto de $\frac{n}{p}$ dados em cestos $B_{i,j}$, contendo os elementos entre o $(j-1)$-ésimo e o j-ésimo elementos da amostra G recebida de P_1, onde $1 \leq j \leq p$.
8. Cada processador P_i envia o cesto $B_{i,j}$ ao processador P_j, onde $1 \leq i, j \leq p$.
9. Cada processador P_i ordena o conjunto de elementos recebido no passo anterior, denominado de R_i, utilizando um algoritmo seqüencial de tempo $O(n \log n)$, e calcula $r_i = \mid R_i \mid$, onde $1 \leq i \leq p$.
10. Cada processador P_i envia r_i para P_1, $1 \leq i \leq p$.
11. P_1 calcula, para cada processador P_j, um vetor A_j, contendo a quantidade e para quais processadores devem ser enviados alguns elementos, sem que a ordem dos elementos seja alterada. Este passo é necessário para distribuir de forma balanceada os elementos pelos processadores.
12. P_1 envia A_j para processador P_j, $1 \leq j \leq p$.
13. Baseado em A_j, cada processador P_j envia alguns de seus elementos para os demais processadores.

fim algoritmo

Teorema 3.1 *O algoritmo* Ordenação_CD *ordena corretamente n elementos armazenados em um CGM/BSP de p processadores, $\frac{n}{p}$ inteiros por processador, $\frac{n}{p} \geq p^2$, usando 6 (2 $\frac{n}{p}$-relações e 4 p^2 relações) rodadas de comunicação, $O(\frac{n}{p})$ de memória local por processador e tempo de computação local $O(\frac{n \log n}{p})$ [2, 7].*

Na Figura 4(a), temos os dados distribuídos entre os processadores com as amostras locais de cada um dos processadores em destaque. Em seguida, cada um dos processadores envia sua amostra para P_1. O vetor com as amostras recebidas e já ordenadas encontra-se na Figura 4(b). O processador P_1 seleciona uma amostra G de p elementos, como mostram as células hachuradas da Figura 4(b). O processador P_1 envia a amostra G para todos os processadores P_i, que dividem seus $\frac{n}{p}$ elementos em p cestos $B_{i,j}$. Cada processador i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq 4$. A Figura 4(c) mostra os cestos recebidos por cada um dos processadores. Cada um dos processadores ordena localmente os elementos recebidos e envia para P_1 a quantidade de inteiros recebida, denominada de r_i.

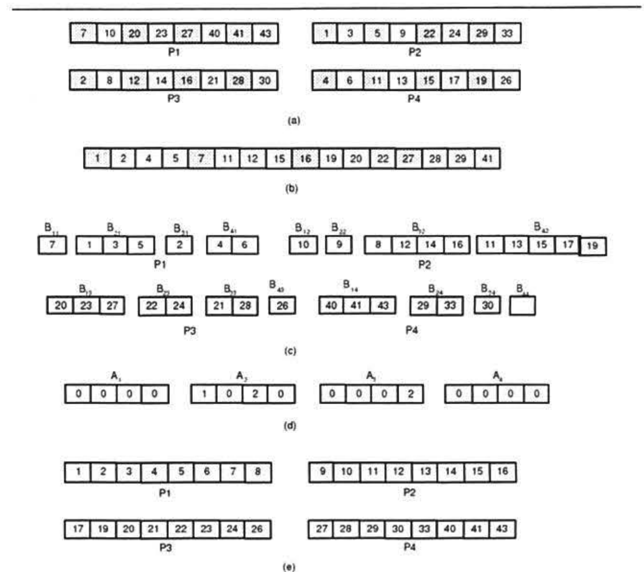

Figura 4. Exemplo de execução do algoritmo Ordenação_CD.

No exemplo da Figura 4, $r_1 = 7$, $r_2 = 11$, $r_3 = 8$, $r_4 = 6$. Baseados nestes valores, o processador P_1 calcula, para cada processador j, um vetor A_j, contendo a quantidade de elementos a serem enviados para cada um dos processadores. Na Figura 4(d), por exemplo, A_2 =(1, 0, 2, 0), ou seja, o processador P_2 deve enviar um elemento para P_1 e dois para P_3. Após esta operação, todos os elementos estão ordenados e cada processador possui exatamente $\frac{n}{p}$ elementos.

3.2. Implementação e Resultados

Nas Figuras 5 e 6 apresentamos, respectivamente, os tempos de execução e *speedup* obtidos através de experimentos do algoritmo *Ordenação_CD* para $1048576 \leq n \leq 8388608$.

Podemos observar que em todos os casos, não importando o valor de n, à medida que aumentamos o número de processadores, o tempo de execução diminui. Além disso, podemos verificar que quanto maior o tamanho da entrada, melhor o desempenho do algoritmo. Isto ocorre, porque o número de rodadas de comunicação é constante, independente dos valores de n e p, e o tempo gasto com computação local em cada processador também é pequeno.

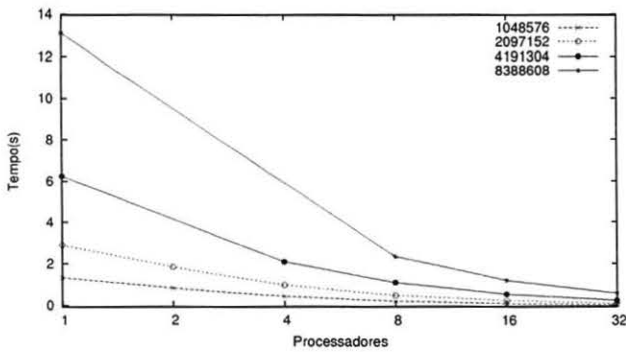

Figura 5. Tempo de execução do algoritmo *Ordenação_CD,* $1048576 \leq n \leq 8388608$.

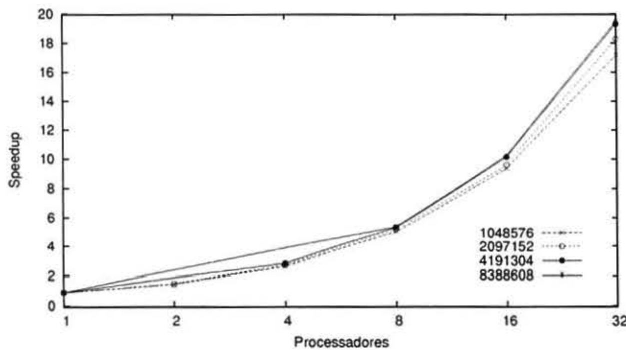

Figura 6. *Speedup* do algoritmo *Ordenação_CD,* $1048576 \leq n \leq 8388608$.

4. Ordenação por Divisão

Nesta seção apresentaremos um algoritmo denominado de *Ordenação_Divisão* que consiste em dividir um conjunto de números em cestos, e distribuir os cestos de forma adequada, para que se possa ordenar n números divididos por p processadores, utilizando $O(1)$ rodadas de comunicação para $\frac{n}{p} \geq p^2$.

Na divisão dos cestos, utilizamos a idéia de calcular um conjunto de divisores, denominados de p-quartis, baseado no cálculo das medianas de um conjunto de elementos. No algoritmo *Ordenação_Divisão* [6], a ser apresentado detalhadamente na Seção 4.1, não há um balanceamento após a ordenação dos dados, devido à forma como os divisores são calculados. Apresentaremos na Seção 4.2 os resultados obtidos com a implementação.

4.1. Descrição do Algoritmo

No algoritmo *Ordenação_Divisão* inicialmente cada processador divide localmente seus $\frac{n}{p}$ números em p subconjuntos, baseados em um conjunto de divisores globais. Este conjunto define p intervalos, que serão utilizados por cada um dos cestos de cada processador. Dessa forma, os elementos do cesto i de um processador estão no mesmo intervalo dos demais elementos do cesto i dos demais processadores.

Por fim, cada processador envia seu i-ésimo cesto para o processador P_i, fazendo com que os elementos do processador P_j sejam menores que os elementos do processador P_k, se $j < k$.

No algoritmo paralelo descrito a seguir, após o envio dos cestos, a quantidade de elementos em cada processador é a mais próxima possível. Para isso, fazemos o cálculo dos divisores globais baseado em um conjunto de divisores locais, calculados separadamente em cada um dos processadores. Em cada processador, os divisores locais, denominados de p-quartis separam os $\frac{n}{p}$ elementos em p cestos de mesmo tamanho.

Definição 4.1 *Dado um conjunto ordenado A de tamanho n, definimos os p-quartis de A como os $p-1$ elementos de índice $\frac{n}{p}, \frac{2n}{p}, \ldots, \frac{(p-1)n}{p}$, que dividem A em p partes de mesmo tamanho.*

Para calcular os divisores de um vetor que particiona o conjunto de elementos em p subconjuntos, utilizamos o algoritmo p-quartis_CGM, descrito a seguir.

Algoritmo: p-quartis_CGM
Entrada: um vetor A com n elementos, divididos pelos p processadores, $\frac{n}{p}$ elementos em cada.
Saída: os divisores globais de A.

1. Cada processador P_i calcula seqüencialmente seus p-quartis através do cálculo das medianas recursivamente, até que $p-1$ elementos sejam encontrados. Denominaremos Q_i os p-quartis calculados por $P_i, 1 \leq i \leq p$.
2. Todos os processadores P_i enviam Q_i para processador P_1, $1 \leq i \leq p$. Denominaremos de Q, os elementos recebidos por P_1.
3. P_1 calcula os p-quartis de Q, obtendo os divisores globais.

fim algoritmo

Teorema 4.1 *O algoritmo* p-quartis_CGM *computa os divisores globais de um conjunto de n elementos divididos em p processadores, $\frac{n}{p} \geq p^2$, em $O(1)$ rodadas de comunicação (1 p-relação) e tempo de computação local $O(\frac{n \log p}{p})$ [7].*

Após calcular o conjunto de divisores, apresentaremos um algoritmo CGM que ordena um conjunto de números dividindo-os em cestos.

Algoritmo: Ordenação_Divisão
Entrada: um vetor A com n elementos, divididos pelos p processadores, $\frac{n}{p}$ elementos em cada.
Saída: todos os elementos de A ordenados, divididos pelos processadores.

1. Compute o conjunto divisor S, utilizando o algoritmo *p-quartis_CGM*.
2. Processador P_1 envia S para todos os processadores.
3. Cada processador P_i, particiona seus elementos em cestos $B_{i,j}$ de acordo com os elementos de S, $1 \leq i, j \leq p$.
4. Cada processador P_i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq p$.
5. Cada processador P_i faz a ordenação dos dados recebidos no passo anterior.

fim algoritmo

Teorema 4.2 *O algoritmo* Ordenação_Divisão *ordena corretamente n inteiros distribuídos por p processadores, utilizando 3 rodadas de comunicação (2 p-relações e 1 $\frac{n}{p}$-relação) e tempo de computação local $O(\frac{n \log n}{p})$ [7].*

Para ilustrar o funcionamento do algoritmo *Ordenação_Divisão*, considere a Figura 7.

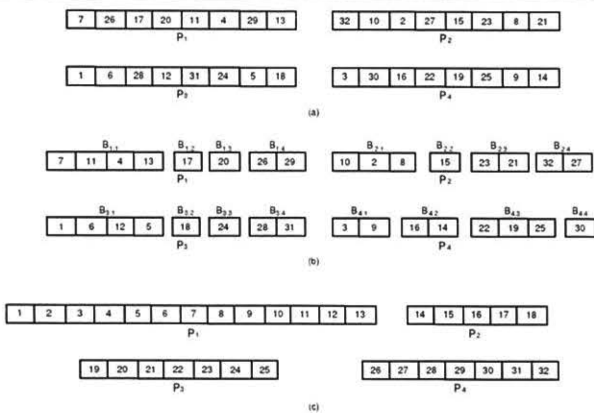

Figura 7. Algoritmo *Ordenação_Divisão*.

Na Figura 7, consideraremos que os divisores já foram calculados e enviados a todos os processadores, ou seja, os Passos 1 e 2 do algoritmo já foram executados.

Baseados nos divisores recebidos de P_1, cada um dos processadores P_i divide seus elementos em p cestos (Figura 7(b)). Em seguida, cada um dos processadores P_i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq p$. Por exemplo, $B_{1,2}$, foi enviado por P_1 ao processador P_2.

Após todos os cestos serem enviados, cada processador ordena localmente os dados recebidos no Passo 4, como mostra a Figura 7(c).

4.2. Implementação e Resultados

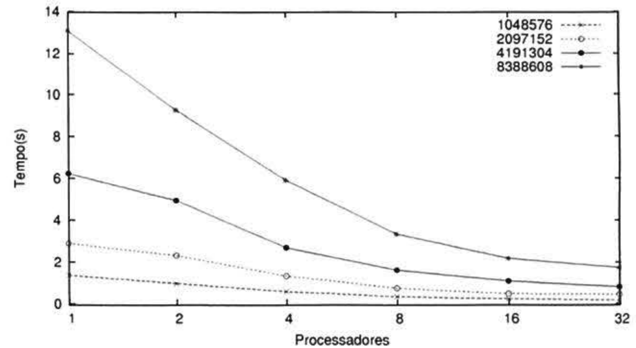

Figura 8. Tempo de execução do algoritmo *Ordenação_Divisão*.

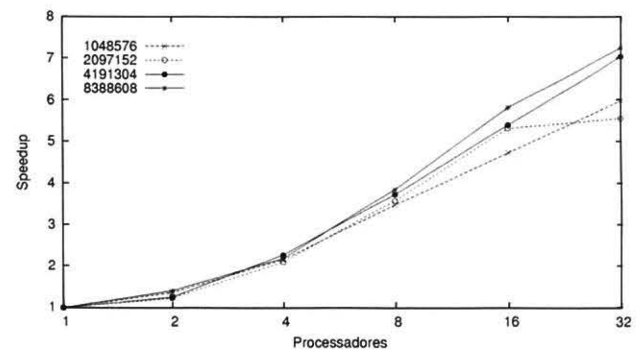

Figura 9. *Speedup* do algoritmo *Ordenação_Divisão*.

Nas Figuras 8 e 9, apresentemos, respectivamente, os tempos de execução e *speedup* obtidos dos experimentos realizados para o algoritmo *Ordenação_Divisão* para $1048576 \leq n \leq 8388608$.

Podemos notar que, ao aumentarmos o número de processadores, o tempo de execução do algoritmo diminui e o

speedup aumenta, ou seja, o desempenho do algoritmo melhora. Isto ocorre pelo fato de termos sempre um número constante de rodadas de comunicação. Assim, ao aumentarmos o número de processadores, temos o mesmo número de rodadas de comunicação, porém o tempo gasto com computação local diminui, pois existem mais processadores realizando a operação de ordenação.

5. Conclusão

Neste trabalho descrevemos três algoritmos de ordenação paralelos, implementados no modelo CGM.

O algoritmo *Ordenação_Bitônica* é um algoritmo que apresenta um ganho significativo em relação ao algoritmo de ordenação bitônica seqüencial. Além disso, neste algoritmo, temos sempre $O(\frac{n}{p})$ elementos em cada processador, o que permite que o mesmo seja facilmente adaptado para ser utilizado em problemas que utilizam ordenação externa. Outra característica interessante deste algoritmo é que a troca de mensagens é realizada sempre entre pares de processadores e cada processador envia e recebe sempre $\frac{n}{p}$ elementos em cada rodada de comunicação.

De acordo com os resultados experimentais apresentados na Seção 3.2, a implementação do algoritmo *Ordenação_CD* apresenta um bom desempenho. Podemos observar que independente do tamanho da entrada, ao aumentarmos o número de processadores, o tempo de execução do algoritmo diminui. Por exemplo, com 32 processadores, o algoritmo *Ordenação_CD* executa 17 vezes mais rápido do que o algoritmo seqüencial para entradas grandes ($1048576 \leq n \leq 8388608$).

O algoritmo *Ordenação_Divisão*, de acordo com os resultados experimentais, apresenta um bom desempenho quando o tamanho da entrada é grande.

Dentre os algoritmos implementados, o que apresentou melhor desempenho de acordo com os resultados experimentais foi o algoritmo *Ordenação_CD*. Mesmo apresentando a mesma complexidade teórica, os algoritmos *Ordenação_CD* e *Ordenação_Divisão* não apresentam o mesmo desempenho na prática. Isto ocorre em conseqüência da forma como o conjunto de divisores é calculado nos dois algoritmos. No algoritmo *Ordenação_CD*, em cada um dos p processadores, os $\frac{n}{p}$ elementos são ordenados para o cálculo dos divisores locais. Conseqüentemente, a divisão dos elementos em cestos, bem como a junção dos cestos recebidos, pode ser implementada de forma mais simples e eficiente que no algoritmo *Ordenação_Divisão*. Por exemplo, como os elementos estão ordenados, basta realizar uma intercalação dos cestos, ao invés de uma ordenação, como ocorre no algoritmo *Ordenação_Divisão*.

Trabalhos futuros incluem a implementação destes algoritmos utilizando o paradigma da orientação a objetos para a construção de uma classe que envolva algoritmos BSP/CGM para ordenação.

Além disso, uma melhoria a ser implementada no algoritmo *Ordenação_Bitônica*, de forma a diminuir a quantidade de informações trocadas entre os processadores, enviando apenas as informações necessárias para a realização da operação de divisão bitônica local. Em conseqüência disso, também conseguiríamos diminuir o tempo de computação local, já que temos menos dados para a realização da operação local.

Em casos onde os elementos sejam inteiros, podemos utilizar algoritmos de tempo linear para as ordenações locais, reduzindo os tempos gastos com computação local.

Também poderia ser incluída como trabalho futuro, a implementação de algoritmos que apresentaram bom desempenho quando descritos em outros modelos, como o *radix sort*, apresentado por Blelloch *et al.* [1].

Referências

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparision of sorting algorithms for the conection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[2] A. Chan and F. Dehne. A note on coarsed grained parallel integer sorting. In *Parallel Processing Letter*, volume 9, pages 533–538, 1999.

[3] E. N. Cáceres, H. Mongelli, and S. W. Song. *Algoritmos Paralelos Usando CGM/PVM: Uma Introdução*, pages 219–278. XXI Congresso da Sociedade Brasileira de Computação, Julho 2001.

[4] 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.

[5] F. Dehne, A. Ferreira, E. N. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarsed grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002.

[6] A. Ferreira. Discrete computing with PC clusters: an algorithmic approach. Tutorial, International School on Advanced Algorithmic Techniques for Parallel Computation with Applications, Setembro 1999.

[7] L. Gonda. Algoritmos BSP/CGM para ordenação. Master's thesis, Universidade Federal de Mato Grosso do Sul, Agosto 2004.

[8] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, 1973.

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

[10] B. Wilkinson and M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.