Algoritmos Paralelos: Ordenação*

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

1 Ordenação

O objetivo da ordenação de números é o de colocar numa determinada ordem numérica (crescente ou decrescente) um conjunto de Números. Caso existam nesse conjunto números duplicados, podemos coloca-los em ordem não crescente ou não decrescente.

O algoritmo de Cole [1] efetua a ordenação de n elementos em tempo paralelo $O(\log n)$ usando n processadores no modelo PRAM CREW. Esse algoritmo foi adaptado para o modelo CGM por Goodrich [4]. No algoritmo proposto por Goodrich [4], os $O(n)$ elementos estão uniformemente distribuídos entre os p processadores e a ordenação dos $O(n)$ elementos é efetuada com tempo de computação local $O(n \log n/p)$ usando $O(\log n/\log(h+1))$ rodadas de comunicação, onde $p < n^{1-\frac{1}{c}}(c \geq 1)$, e $h = \Theta(\frac{n}{p})$, ou seja, com computação local ótima e $O(1)$ rodadas de comunicação quando $\frac{n}{p} \geq p$.

Embora esses algoritmos sejam ótimos, do ponto de vista teórico, suas implementações utilizam uma grande quantidade de detalhes. Vamos agora descrever vários algoritmos de ordenação que são mais simples. Vamos apresentar quatro algoritmos paralelos de ordenação: Ordenação por Inserção,

*Este material é para o uso exclusivo do Curso de Algoritmos Paralelos do DCT da UFMS. Parte desse material refere-se aos livros Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers de Barry Wilkinson e Michael Allen, Parallel Programming with MPI de Peter Pacheco, além de Texto e Algoritmos elaborados por Edson Norberto Cáceres

Ordenação por Buckets, Ordenação Bitônica e Ordenação Split.Vamos discutir os detalhes da implementação desses algoritmos no modelo BSP/CGM.

1.1 Ordenação por Inserção

Inicialmente vamos apresentar um algoritmo paralelo BSP/CGM com $O(p)$ rodadas de comunicação e $O(\frac{n^2}{p})$ computação local para ordenar um conjunto de n números utilizando p processadores. O algoritmo que vamos apresentar é uma versão paralela do algoritmo seqüencial de ordenação por inserção. A técnica que vamos usar para implementar esse algoritmo é conhecida como computação *pipeline*.

A idéia do algoritmo é a seguinte: Inicialmente, na rodada 0, o processador P_0 lê n/p números. Usando o algoritmo de ordenação por inserção, o processador P_0 ordena os n/p números. Na rodada 1, o processador P_0 lê mais n/p números e, novamente usando o algoritmo seqüencial de ordenação por inserção, ordena os $2n/p$ números. Após ordena-los, o processador P_0 retém os n/p menores e envia para P_1 os n/p maiores. O processador P_1 recebe os n/p números enviados por P_0. Na próxima rodada, a 2, o processador P_0 lê mais n/p números e, novamente usando o algoritmo seqüencial de ordenação por inserção, ordena os $2n/p$ números. Após ordena-los, o processador P_0 retém os n/p menores e envia para P_1 os n/p maiores. O processador P_1 recebe os n/p números enviados por P_0 e efetua a ordenação desses números em conjunto com os outros n/p números recebidos na rodada anterior. Retém os n/p menores e envia os n/p maiores para P_2. Temos que na rodada i, o processador P_0 lê n/p novos números e os processadores $P_1, P_2, \ldots, P_j$ recebem n/p novos números dos processadores P_{j-1}, $1 \leq j \leq i-1$, efetuam a ordenação dos $2n/p$ números, retêm os n/p menores e enviam os n/p maiores para P_{j+1}. O processador P_{j+1} recebe os n/p números.

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

Usando este escalonamento (Figura 1), podemos ver que na rodada 0, somente o processador P_0 trabalha. Na segunda rodada, os processadores P_0 e P_1 trabalham. É fácil de ver que na rodada k, somente os processadores P_i trabalham, onde $1 \leq i \leq k$.

Algoritmo 1 Ordenação por Inserção
Entrada: (1) O número p de processadores; (2) O número i do processador, onde the $0 \leq i \leq p-1$; e (3) Um conjunto de n números distintos, $S =$

$\{s_0, s_1, \ldots, s_{n-1}\}$.
Saída: $S' = \{s'_0, s'_1, \ldots, s'_{n-1}\}$, onde $s'_{i-1} < s'_i$.
(1) **if** $i = 0$ **then**
  (1.1) `leia` $S_0 = \{s_0, s_1, \ldots, s_{n/p-1}\}$`;`
  (1.2) $S'_0 \leftarrow$ `Ordene(`S_0`);`
  else $S'_0 \leftarrow \emptyset$
(2) **for** $1 \leq k \leq p - 1$
  (2.1) **if** $i = 0$ **then**
    (2.1.1) `leia` $S_k = \{s_{kn/p}, s_{kn/p+1}, \ldots, s_{(k+1)n/p-1}\}$`;`
    (2.1.2) $S'_k \leftarrow$ `Ordene(`$S'_{k-1} \cup S_k$`);`
    (2.1.3) $R_k \leftarrow s'_{n/p}, s'_{n/p+1}, \ldots, s'_{2n/p-1}$;
    (2.1.4) $S'_k \leftarrow s'_0, s'_1, \ldots, s'_{n/p-1}$;
    (2.1.5) **send**`(`R_k`,`P_1`);`
  (2.2) **if** $i \neq 0$ **then**
    (2.2.1) **receive**`(`R_k, P_{i-1}`);`
    (2.2.2) $S'_k \leftarrow$ `Ordene(`$S'_{k-1} \cup R_k$`);`
    (2.2.3) $R_k \leftarrow s'_{n/p}, s'_{n/p+1}, \ldots, s'_{2n/p-1}$;
    (2.2.4) $S'_k \leftarrow s'_0, s'_1, \ldots, s'_{n/p-1}$;
  (2.3) **if** $i \neq p - 1$ **then**
    (2.3.1) **send**`(`R_k`,`P_{i+1}`);`
— Fim do Algoritmo —

1.2 Corretude e Complexidade

2 Ordenação por *Buckets*

O algoritmo de ordenação por *buckets* (*bucket sort*) não é baseado em comparação e troca, sendo um algoritmo que naturalmente é um método de particionamento. Contudo a ordenação por buckets somente funciona bem se o conjunto de números está uniformemente distribuído em um intervalo conhecido, digamos 0 a $L - 1$. Este intervalo é dividido em m regiões iguais 0 a $L/m - 1$, L/m a $2L/m - 1$, ... e um *bucket* é atribuído para armazenar os números que pertencem a cada região. Desta forma, temos m buckets. A ordenação é feita colocando cada número no bucket apropriado. O algoritmo pode ser usado com 1 bucket para cada número (i.e. m=n). No caso em que começamos com 2 buckets e através do método de divisão e conquista, dividirmos continuamente os buckets em buckets menores até que cada bucket armazene um único número, temos um método semelhante ao *quicksort*, exceto pelo fato de que no *quicksort* as regiões são definidas pelo *pivots* [2]. Usando um número limitado m de buckets, temos que a complexidade do algoritmo seqüencial é $O(n \log \frac{n}{m})$. Caso $n = km$, para alguma constante k, temos que a complexidade é $O(m)$.

2.1 Algoritmo Paralelo

Claramente a ordenação por buckets pode ser paralelizada. Consideremos um conjunto S com n números, e o modelo CGM com p processadores. Vamos considerar p buckets com n/p elementos cada, atribuindo um bucket para cada processador P_i.

Vamos agora descrever a idéia do algoritmo. Inicialmente o conjunto S de n elementos é dividido em p subconjuntos S_i de n/p elementos cada. Cada processador P_i recebe um subconjunto S_i. Dados o $MIN = \min\{s_0, s_1, \ldots, s_{n-1}\}$ e $MAX = \{s_0, s_1, \ldots, s_{n-1}\}$ e $B = (MAX - MIN)/p$, o conjunto S é dividido em p regiões MIN a $MIN + B$, $MIN + B + 1$ a $MIN + 2B$, …, $MIN + (p-1)B$ a MAX,que são os buckets B_i^j. Cada processador P_i distribui os elementos do subconjunto S_i entre os p buckets B_i^j, $0 \leq j \leq p-1$. Cada processador P_i envia o conteúdo do bucket B_i^j para o processador P_j, e cada processador P_i recebe $p-1$ buckets $B_0^i, B_1^i, \ldots, B_{i-1}^i, B_{i+1}^i, \ldots, B_{p-1}^i$. Ao final dessa rodada, cada processador P_i estará armazenando os números de S que estão na região $MIN + iB + 1$ a $MIN + (i+1)B$. A seguir, cada processador P_i efetua a ordenação do seu bucket

$$\bigcup_{j=0}^{p-1} B_j^i$$

.

Algoritmo 2 Bucket Sort
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os p processadores (n/p elementos por processador).
Saída: Todos elementos ordenados dentro de cada processador e por processador, ou seja, se $i < j$, temos que os elementos em p_i são menores que os elementos pertencentes a p_j.
(1) `Compute` MAX `e` MIN `do conjunto` A `e` B;
(2) `Compute um conjunto divisor` $D = \{MIN = d_1, d_2, \ldots, d_{p-1} = MIN + (p-1)B\}$;

(3) `Particionar os elementos de` p_i `em buckets` B_i^j `de acordo com` D`;`
(4) `send(`B_i^j`,`p_j`);`
(5) `receive(`B_j^i`,`p_j`);`
(6) `Ordene` $\bigcup_{j=0}^{p-1} B_j^i = B_0^i \cup B_1^i \ldots B_{p-1}^i$`;`
— Fim do Algoritmo —

2.1.1 Corretude e Complexidade

O algoritmo de ordenação por bucktes ordena um conjunto de n elementos em $O(1)$ rodadas de comunicação efetuando $O(\frac{n}{p}\log\frac{n}{p})$ computação local.

3 Ordenação Bitônica

A base da **ordenação bitônica** é o conceito de seqüência **bitônica**, uma lista com propriedades específicas que serão utilizadas no algoritmo de ordenação. Uma **seqüência monotônia crescente (decrescente)** é uma seqüencia crescente (decrescente) de números. Uma **seqüência bitônica** é composta de duas seqüências monotônicas, uma crescente (decrescente) e outra decrescente (crescente). Formalmente, uma **seqüência bitônica** é uma seqüência de números $(s_0, s_1, \ldots, s_{n-1})$, a qual cresce (decresce) monotonicamente, atinge um único máximo (mínimo), e então decresce (cresce) monotonicamente; por exemplo,

$$s - 0 < s_1 < \ldots < s_{i-1} < s_i > s_{i+1} > s_{i+2} > \ldots > s_{n-2} > s_{n-1}, 0 \leq i < n$$

A seqüência também é bitônica se existe um deslocamento cíclico σ de $(0, 1, \ldots, n-1)$ tal que a seqüência $(s_{\sigma(0)}, s_{\sigma(1)}, \ldots, s_{\sigma(n-1)})$ satisfaça a condição anterior. Notamos que seqüências bitônicas podem ser formadas pela concatenação de duas listas ordenadas, uma em ordem crescente (decrescente) e outra em ordem decrescente (crescente).

Exemplo 1 *Exemplo*

Uma característica especial das seqüências bitônicas é que se executarmos uma operação de comparação e troca em s_i com $s_{i+n/2}$ para todo $0 \leq i < n$, obtemos duas seqüências bitônicas, onde os números de uma seqüência são menores que os números da outra seqüência. Podemos formalizar essa propriedade da seguinte forma:

$$S_{min} = (\min\{s_0, s_{\frac{n}{2}}\}, \min\{s_1, s_{\frac{n}{2}+1}\}, \ldots, \min\{s_{\frac{n}{2}-1}, s_{n-1}\}$$

e

$$S_{max} = (\max\{s_0, s_{\frac{n}{2}}\}, \max\{s_1, s_{\frac{n}{2}+1}\}, \ldots, \max\{s_{\frac{n}{2}-1}, s_{n-1}\}$$

Essa característica *especial* das seqüências bitônicas é denominado *split* bitônico.

Exemplo 2 *Considere a seguinte seqüência bitônica*

2 5 7 9 8 4 3 0

e executando comparação e troca, temos que $S_{min} = (2, 4, 3, 0)$ *e* $S_{max} = (8, 5, 7, 9)$.

A operação compare e troque move os menores números de cada par para S_{min} e os números maiores para S_{max}. Usando o fato de que todos elementos de S_{min} são menores que todos elementos de S_{max}, e que ambas as seqüências são bitônicas, podemos dividir, recursivamente, S_{min} e S_{max} até obtermos seqüências de tamanho 1. Quando isso ocorrer, obteremos a seqüência ordenada.

A Tabela 3 ilustra o procedimento de dividir sucessivamente uma dada seqüência para obter uma seqüência ordenada.

Passo	*Elementos*							
0	2	5	7	9	8	4	3	0
1	2	4	3	0	8	5	7	9
2	2	0	3	4	7	5	8	9
3	0	2	3	4	5	7	8	9

Com a resultado anterior, temos que dada uma seqüência bitônica é bastante simples obter uma seqüência ordenada. Falta-nos obter uma forma de que dada uma seqüência qualquer, como torna-la bitônica.

Iniciamos com seqüências formadas com pares de números adjacentes no conjunto a ser ordenado. Observamos que qualquer seqüência contendo dois elementos é bitônica, de fato, qualquer seqüência contendo dois elementos é monotônica.

Para ilustrar a idéia principal, vamos supor inicialmente que nossa seqüência de números não ordenados possui quatro elementos, $S = (s_0, s_1, s_2, s_3)$. Para que S seja bitônica, não é suficiente que as duas subseqüências (s_0, s_1) e (s_2, s_3) sejam monotônicas. Necessitamos que a primeira seja crescente (decrescente) e a segunda seja decrescente (crescente). Isso é fácil de ser obtido, para isso basta comparar os dois elementos de cada seqüência e coloca-los na forma apropriada.

Basta agora obtermos uma forma de generalizar essa idéia. Em uma seqüência de oito elementos, $S = (s_0, s_1, \ldots, s_7)$, podemos utilizar o método anterior e converter S em duas subseqüências bitônicas, (s_0, s_1, s_2, s_3) e (s_4, s_5, s_6, s_7). Além disso podemos supor que $s_0 \leq s_1$, $s_2 \geq s_3$, $s_4 \leq s_5$ e $s_6 \geq s_7$. Basta agora transformar S em bitônica. Temos que a aplicação sucessiva de *splits* bitônicos converte uma seqüência bitônica em uma seqüência ordenada. Desta forma, podemos transformar a subseqüência (s_0, s_1, s_2, s_3) em crescente e mudando a forma do *split* bitônico, colocando os máximos na primeira metade e os mínimos na segunda, podemos transformar (s_4, s_5, s_6, s_7) em decrescente.

Passo	*Elementos*							
0	8	3	4	7	9	2	0	5
1	3	8	7	4	2	9	5	0
2	3	4	7	8	5	9	2	0
3	3	4	7	8	9	5	2	0

3.1 Corretude e Complexidade

Teorema 1 *O algoritmo de ordenação bitônica ordena corretamente um conjunto S de n números em tempo $O(\log^2 n)$.*

Demonstração: □

3.2 Algoritmo Paralelo

Da mesma forma que no caso seqüencial, vamos assumir que o número de elementos n do conjunto S a serem ordenado seja uma potência de dois. Assumimos também que o número de processadores p seja uma potência de dois. A cada rodada do algoritmo, cada processador conterá um subconjunto do conjunto global contendo $\frac{n}{p}$ elementos.

Vamos analisar o caso em que $p = 4$. Podemos iniciar com a utilização de um algoritmo de ordenação local para converter a subseqüência local em crescente ou decrescente. A ordenação será determinada pela paridade do número do processador. Processadores rotulados com números pares utilizarão ordenação crescente e processadores rotulados com números ímpares utilizarão ordenação decrescente. Desta forma, cada par de processadores, $(0, 1)$ e $(2, 3)$, em conjunto armazenam seqüências bitônicas, e se os processadores executarem splits bitônicos apropriados, cada par conterá uma seqüência monotônica e a seqüência global será bitônica.

Observamos que é possível executar um split bitônico em uma seqüência em uma seqüência totalmente distribuída através da execução de um split bitônico na seqüência compartilhada pelos processadores 0 e 2 e executando um split bitônico na seqüência compartilhada pelos processadores 1 e 3.

Exemplo 3 *Seja a seqüência bitônica*

$$(9, 12, 16, 23, 26, 39, 42, 61, 43, 17, 14, 13, 12, 7, 6, 5)$$

se efetuarmos um split bitônico crescente e distribuir o resultado entre os 4 processadores teremos:

Processador	Elementos			
0	*9*	*12*	*14*	*13*
1	*12*	*7*	*6*	*5*
2	*43*	*17*	*16*	*23*
3	*26*	*39*	*42*	*61*

Por outro lado se distribuirmos o conjunto bitônico inicial (antes de efetuar o split bitônico) entre os 4 processadores teremos:

Processador	Elementos			
0	*9*	*12*	*16*	*23*
1	*26*	*39*	*42*	*61*
2	*43*	*17*	*14*	*13*
3	*12*	*7*	*6*	*5*

Fazendo um split bitônico na subseqüência armazenada no processadores P_0 *e* P_2*,* $(9, 12, 16, 23, 43, 17, 14, 13)$ *e armazenar o resultado* S_{min} *em* P_0 *e* S_{max} *em* P_2*. De maneira análoga, fazemos um split bitônico nos elementos armazenados nos processadores* P_1 *e* P_3 *e armazenamos o resultado de* S_{min} *em* P_1 *e* S_{max} *em* P_3*. Ao final dessas duas operações, os processadores* P_0*,* P_1*,* P_2 *e* P_3 *estarão armazenado os mesmos elementos que a primeira tabela do exemplo. Ou seja um split bitônico em uma seqüência global tem o mesmo efeito que dois splits bitônicos independentes: um efetuado na subseqüência compartilhada pelos processadores* P_0 *e* P_2*, e outro na subseqüência compartilhada pelos processadores* P_1 *e* P_3*. A ordenação é obtida com um split bitônico nas subseqüências armazenadas pelos processadores* P_0 *com* P_1 *e* P_1 *com* P_2*.*

A generalização de quatro processadores para o caso com $p = 2^d$ processadores não é difícil de obter. No caso de oito processadores, inciciamos com um split bitônico nos pares de processadores $(0, 4)$, $(1, 5)$, $(2, 6)$ e $(3, 7)$. O próxima conjunto de splits bitônicos será aplicado nos pares $(0, 2)$, $(1, 3)$, $(4, 6)$ e $(5, 7)$. Finalmente aplicamos um split bitônico nos pares $(0, 1)$, $(2, 3)$, $(4, 5)$ e $(6, 7)$

3.2.1 Corretude e Complexidade

4 Split Sort

O algoritmo de ordenação por buckets exige que o conjunto de números (chaves) esteja uniformemente distribuído. Vamos introduzir uma estratégia para obter o conjunto separador, de tal forma que cada bucket armazene $O(n/p)$ elementos, e não haja a exigência de que o conjunto de números a ser ordenado esteja uniformemente distribuído. O algoritmo que agora vamos descrever é o *split-sort*. Esse algoritmo é um pouco menos escalável, mas ainda é ótimo e com muito poucas rodadas de comunicação, para $\frac{n}{p} \geq p$.

Algoritmo 3 CGM Split Sort
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os p processadores (n/p elementos por processador).
Saída: Todos elementos ordenados dentro de cada processador e por processador, ou seja, se $i < j$, temos que os elementos em p_i são menores que os elementos pertencentes a p_j.
(1) `Compute um conjunto divisor` $S = \{s_1, s_2, \ldots, s_{p-1}\}$;
(2) `broadcast(S,`p_i`);`
(3) `Particionar os elementos de` p_i `em buckets` B_j^i `de acordo com` S;
(4) `send(`B_j^i`,`p_j`);`
(5) `Ordene` $B_i^k = B_i^0 \cup B_i^1 \ldots B_i^{p-1}$;
— Fim do Algoritmo —

É fácil verificar que o Algoritmo 3 ordena qualquer entrada, visto que não foi efetuado nenhuma restrição ao tamanho dos *buckets*. Como no modelo CGM temos que cada processador tem $O(n/p)$ memória local, devemos escolher cuidadosamente o conjunto S, pois isso influenciará no tamanho dos *buckets*.

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

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

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

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

Algoritmo 4 p-quartis Sequencial
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$ o número de quartis.
Saída: O conjunto A dividido em p-quartis.
(1) `Compute a mediana de` A`;`
(2) `Usando a mediana, divida` A `em dois conjuntos` A_1 `e` A_2`;`
(3) `Aplique o algoritmo recursivamente, até que` $p-1$ `splitters sejam encontrados;`
— Fim do Algoritmo —

Usamos o Algoritmo 4 no algoritmo CGM para computar S.

Algoritmo 5 Conjunto Divisor S CGM
Entrada: (1) Um vetor A com n elementos. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os p processadores (n/p elementos por processador).
Saída: O conjunto A dividido em p-quartis.
(1) $Q_i \leftarrow p$`-quartis(`A_i`);`
(2) `send(`Q_i`,`p_0`);`
(3) **se** $i = 0$ **então** `S` $\leftarrow$ `Ordena(`$Q_0 \cup Q_1 \ldots Q_{p-1}$`;`
(4) **se** $i = 0$ **então** `S` $\leftarrow$ `p-quartis(Q)`
— Fim do Algoritmo —

4.1 Corretude e Complexidade

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

Demonstração: □

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

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

Referências

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

[2] T.H. Cormen and C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

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

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

[5] P. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

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