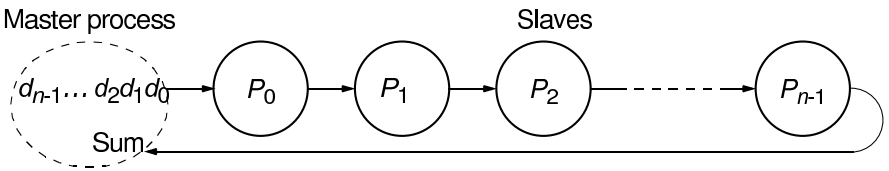
Algoritmos Paralelos de Ordenação - MAC 5705

Baseados no texto JAI/SBC2001 de Cáceres, Mongelli e Song e alguns slides de Wilkinson and Allen

Edson Cáceres e Siang Wun Song
UFMS e USP

Slide 171

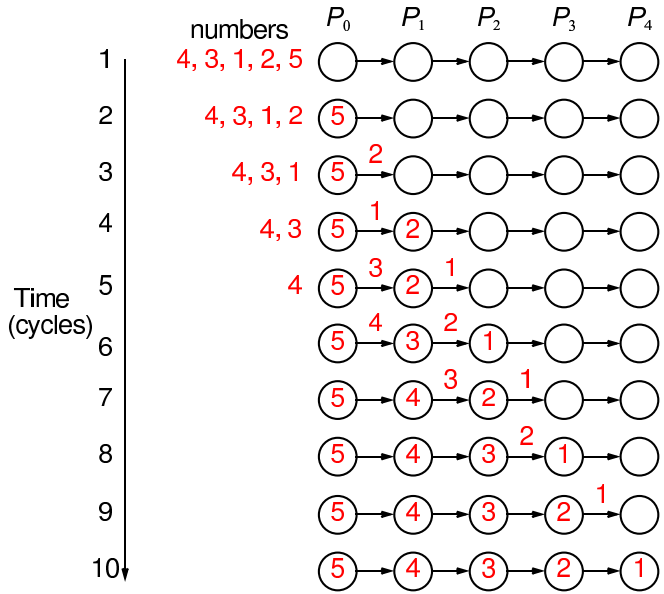
Pipelined addition numbers with a master process and ring configuration



Slide 172

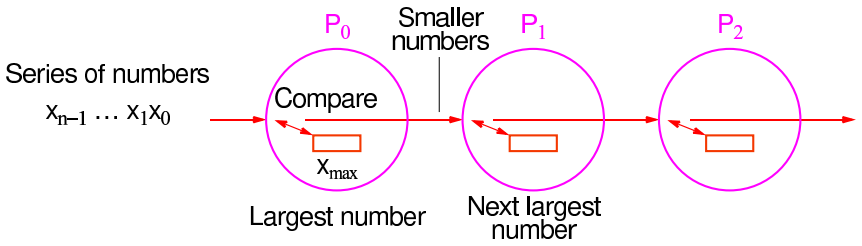
Sorting Numbers

A parallel version of *insertion sort*.



Slide 173

Pipeline for sorting using insertion sort



Type 2 pipeline computation

Slide 174

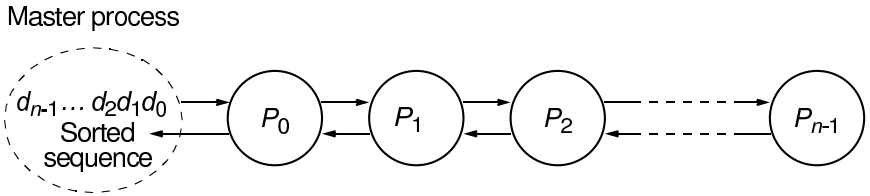
The basic algorithm for process P_i is

```
recv(&number, P_i-1);
if (number > x) {
  send(&x, P_i+1);
  x = number;
} else send(&number, P_i+1);
```

With n numbers, how many the ith process is to accept is known; it is given by $n - i$. How many to pass onward is also known; it is given by $n - i - 1$ since one of the numbers received is not passed onward. Hence, a simple loop could be used.

Slide 175

Insertion sort with results returned to the master process using a bidirectional line configuration



Algoritmo CGM para Ordenação com Inserção

Input: (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, $S = \{s_0, s_1, \ldots, s_{n-1}\}$.
Output: $S' = \{s'_0, s'_1, \ldots, s'_{n-1}\}$, onde $s'_{i-1} < s'_i$.

1: **if** $i = 0$ **then**
2: `leia` $S_0 = \{s_0, s_1, \ldots, s_{n/p-1}\}$;
3: $S'_0 \leftarrow$ `Ordene`(S_0);
4: **else**
5: $S'_0 \leftarrow \emptyset$
6: **end if**
7: **for** $1 \leq k \leq p-1$ **do**
8: **if** $i = 0$ **then**
9: `leia` $S_k = \{s_{kn/p}, s_{kn/p+1}, \ldots, s_{(k+1)n/p-1}\}$;
10: $S'_k \leftarrow$ `Ordene`$(S'_{k-1} \cup S_k)$;

11: $R_k \leftarrow s'_{n/p}, s'_{n/p+1}, \ldots, s'_{2n/p-1}$;
12: $S'_k \leftarrow s'_0, s'_1, \ldots, s'_{n/p-1}$;
13: **send**(R_k, P_1);
14: **end if**
15: **if** $i \neq 0$ **then**
16: **receive**(R_k, P_{i-1});
17: $S'_k \leftarrow$ `Ordene`$(S'_{k-1} \cup R_k)$;
18: $R_k \leftarrow s'_{n/p}, s'_{n/p+1}, \ldots, s'_{2n/p-1}$;
19: $S'_k \leftarrow s'_0, s'_1, \ldots, s'_{n/p-1}$;
20: **end if**
21: **if** $i \neq p-1$ **then**
22: **send**(R_k, P_{i+1});
23: **end if**
24: **end for**

Algoritmo CGM para Ordenação com Inserção - Comentários

- **Entrada de n/p números cada vez, ao invés de um número por processador.**
- **Requer $O(p)$ rodadas de comunicação.**
- **Computação local em cada rodada: ordenar n/p números.**
- **Portanto $O(\frac{n}{p}\log\frac{n}{p})$=$O(\frac{n\log n}{p})$ (em cada rodada).**
- **A computação local total (considerando $O(p)$ rodadas) é $O(n\log n)$ (igual à complexidade seqüencial).**

Características da computação "pipeline"

- **A ordenação por inserção é um exemplo de computação sistólica ou "pipeline".**
- **Computação prossegue em *frente de ondas*.**
- **É interessante por levar em conta a Entrada e Saída dos dados.**
- **Tem a vantagem de exigir comunicação com poucos vizinhos.**
- **No início e no final (quando o pipeline está sendo preenchido) poucos processadores trabalham.**

Ordenação por Partição ou "Bucket Sort"

- **Não é baseado no paradigma de *comparar* e *trocar de posição*.**
- **Usa uma operação importante: *divisão*.**
- **Supõe que números a serem ordenados estão uniformente distribuídos dentro de um intervalo de 0 a $a-1$.**

Idéia do Bucket Sort

Particionar os números (uniformemente distribuídos em $[0, a-1]$) em m intervalos (buckets).

Os buckets são portanto de

- **de 0 a $\frac{a}{m} - 1$,**
- **de $\frac{a}{m}$ a $\frac{2a}{m} - 1$,**
- **de $\frac{2a}{m}$ a $\frac{3a}{m} - 1$, etc.**

Com a suposição de distribuição uniforme no intervalo $[0, a-1]$, a quantidade de números dentro de cada bucket será aproximadamente igual.

Cada partição é então ordenada. (Novamente sub-buckets podem ser usados, dentro do paradigma de divisão e conquista.)

Como Particionar em Buckets

Para cada número deve-se decidir a que bucket ele pertence.

Isso pode ser feito com comparações do número com os extremos de cada bucket até achar o bucket correto.

Mas o melhor é usar uma divisão.

Sejam m buckets. O número x será colocado no bucket $\lfloor \frac{x}{a/m} \rfloor$.

Se m é uma potência de 2, e.g. $m = 2^k$, então a divisão resume em escolher os k bits mais significativos do número binário x.

Por exemplo: $m = 2^3 = 8$. O número $x = 1100101$ é colocado no bucket 110 (3 bits mais significativos de x).

Bucket Sort Seqüencial

Dados n números a serem ordenados, o particionamento em buckets leva tempo $O(n)$.

Cada ordenação do bucket leva tempo $O(\frac{n}{m} \log \frac{n}{m})$.

Portanto para ordenar todos os m buckets: $O(m\frac{n}{m} \log \frac{n}{m}) = O(n \log \frac{n}{m})$.

O tempo total será $O(n + n \log \frac{n}{m})$.

Se fizermos $k = \frac{n}{m}$ = constante, então o Bucket Sort leva tempo $O(n)$.

Lembrete: é necessário ter distribuição uniforme dos números e o uso da divisão.

Bucket Sort Paralelo

Considere p processadores cada um com memória local $O(n/p)$.

Usamos $m = p$ buckets correspondentes aos p processadores.

Supondo a distribuição uniforme dos números no intervalo 0 a $a-1$, temos aprox. n/p números em cada bucket, portanto cada bucket cabe num processador.

Bucket i corresponde ao processador i, $0 \leq i \leq p-1$.

Bucket Sort

Input: **(1) Vetor A de n elems. (2) p processadores $p_0, p_1, \ldots, p_{p-1}$. (3) Elementos de A distribuídos entre os p processadores (n/p elem. por processador).**

Output: **Todos elementos ordenados dentro de cada processador e por processador, i.e. se $i < j$, então os elementos em p_i são menores que os de p_j.**

1: Compute MAX e MIN de A; $B = (MAX - MIN)/p$;
2: Compute um conjunto divisor $D = \{MIN + B, MIN + 2B, \ldots, MIN + (p-1)B\}$;
3: Particionar os elementos de p_i em buckets B_i^j de acord com D;
4: send(B_i^j,p_j);
5: receive(B_j^i,p_j);
6: Ordene $\bigcup_{j=0}^{p-1} B_j^i = B_0^i \cup B_1^i \ldots B_{p-1}^i$;

Algoritmo CGM Bucket Sort

Cada processador tem inicialmente n/p números. Cada processador determina os buckets (0 a $p-1$) de seus números.

Cada processador i envia os seus números do bucket j ao processador j. Recebe por sua vez de outros processadores os números do bucket i.

Cada processador ordena seus números. (O processador 0 terá os menores números, o processador 1 terá os próximos menores, etc.)

- **Rodada de comunicação: constante.**
- **Computação local: $O(\frac{n}{p}\log\frac{n}{p}) = O(\frac{n\log n}{p})$.**
- **Lembre-se da suposição de distribuição uniforme da entrada.**

Ordenação Bitônica

Seqüência bitônica: uma seqüência de números $(s_0, s_1, \ldots, s_{n-1})$ que cresce (decresce) monotonicamente, atinge um único máximo (mínimo), e então decresce (cresce) monotonicamente:

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

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.

Obs: seqüências bitônicas podem ser formadas pela concatenação de uma seq. em ordem crescente (decrescente) e outra em ordem decrescente (crescente).

Exemplo:

2	**5**	**7**	**9**	**8**	**4**	**3**	**0**

Propriedade de "Split" Bitônico

Dada uma seqüência bitônica de números distintos $(s_0, s_1, \ldots, s_{n-1})$**, então as seqüências**
$S_{min} = (\min\{s_0, s_{\frac{n}{2}}\}, \min\{s_1, s_{\frac{n}{2}+1}\}, \ldots, \min\{s_{\frac{n}{2}-1}, s_{n-1}\}$ **e**
$S_{max} = (\max\{s_0, s_{\frac{n}{2}}\}, \max\{s_1, s_{\frac{n}{2}+1}\}, \ldots, \max\{s_{\frac{n}{2}-1}, s_{n-1}\}$
são também bitônicas e, mais ainda, todos os elementos de S_{min} **são menores que todos os elementos de** S_{max}**.**

Exemplo: a seqüência bitônica

2 5 7 9 8 4 3 0
↑ ↑

produz
$S_{min} = (2, 4, 3, 0)$ **e**
$S_{max} = (8, 5, 7, 9)$**.**

Ordenação de uma Seqüência Bitônica

Podemos ordenar uma seqüência bitônica

$$(s_0, s_1, \ldots, s_{n-1})$$

usando sucessivos splits bitônicos. Exemplo:

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

- **Isso funciona para ordenar uma seqüência bitônica.**
- **Pergunta: para ordenar uma seqüência de números quaisquer (não bitônica), como transformá-la em seqüência bitônica?**

Transformar uma Seqüência Qualquer em Bitônica

Seja uma seqüência de 8 números $(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7)$.

- **Ordene pares de números e produza**
  (s'_0, s'_1) crescente e (s'_2, s'_3) decrescente;
  (s'_4, s'_5) crescente e (s'_6, s'_7) decrescente.
- **Ordene a seqüência bitônica (s'_0, s'_1, s'_2, s'_3) em ordem crescente. Ordene a seqüência bitônica (s'_4, s'_5, s'_6, s'_7) em ordem decrescente**

Temos então uma seqüência bitônica de 8 elementos:

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**

Complexidade da Ordenação Bitônica

A ordenação bitônica de uma seqüência qualquer de n números leva tempo $O(\log^2 n)$.

Algoritmo CGM: a ordenação bitônica não é muito adequada para implementar no CGM.

Inicialmente cada processador pode ordenar seqüencialmente seus n/p números. Para implementar a ordenação bitônica são necessárias $O(\log^2 p)$ rodadas de comunicação. (Conferir isso.)


communications
of the
acm
In this issue:
Candidates
for ACM Election

Split Sort ou Ordenação usando Separadores

Bucket Sort precisa da hipótese da distribuição uniforme dos números de entrada. Senão cada bucket teria quantidade diferentes de números.

Split Sort funciona para qualquer distribuição de números, pois a construção dos buckets usa os splits ou separadores.

A mediana é um separador para dividir os números em duas partições, os menores a ela e os maiores a ela.

De modo geral, temos os p-quartis (ou p-splitters ou p-separadores).

Os p-separadores de um conjunto A de tamanho n são os $p-1$ elementos que dividem A em p partições de igual tamanho de n/p números cada.

Split Sort CGM

Input: (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).

Output: 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 acord com S;
4: send(B_j^i,p_j);
5: Ordene $B_i^k = B_i^0 \cup B_i^1 \ldots B_i^{p-1}$

Algoritmo Seqüencial para Obter Separadores

Os separdores (ou p-quartis) podem ser computados seqüencialmente em tempo $O(n \log p)$.

Algoritmo p-quartis:

Input: (1) Um vetor A com n elementos. (2) o número p (número de partições desejadas).

Output: 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;`

Algoritmo Paralelo para Obter Separadores

Algoritmo CGM para Obtenção do Conjunto Divisor S:

Input: **(1) Vetor A com n elementos. (2) p processadores $p_0, \ldots, p_{p-1}$. (3) Cada processador i armazena A_i com n/p elementos de A.**

Output: **O conjunto A dividido (aproximadamente?) em p-quartis.**

1: $Q_i \leftarrow$ `p-quartis(`A_i`);` **(usa o algoritmo sequencial)**
2: `send(`Q_i`,`p_0`);`
3: **if** $i = 0$ **then**
4: `Q` $\leftarrow$ `Ordena(`$Q_0 \cup Q_1 \ldots Q_{p-1}$`);`
5: **end if**
6: **if** $i = 0$ **then**
7: `S` $\leftarrow$ `p-quartis(Q)`
8: **end if**

Complexidade

Teorema: 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 $\frac{n}{p} \geq p^2$.

Radix Sort para Ordenação de Inteiros

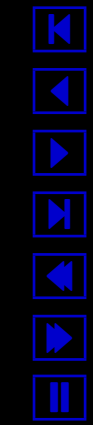
$a_0, a_1, \ldots, a_{n-1}$ **inteiros no intervalo** $[0, m-1]$**.**

Se m **não é muito grande, use** m **buckets, um bucket para cada inteiro de 0 a** $m-1$**.**

Distribui os números, colocando o número i **no bucket** i**.**

Concatene todos os buckets e temos a ordenação feita.

Obs: a quantidade de números em cada bucket pode variar. Note que quando mais do que um número vai para um mesmo bucket, não é necessário colocar esses números iguais no bucket. Basta a quantidade desses números que são todos iguais.



Radix Sort para m grande

Se m é grande, pode usar radix sort por partes.

Considere inteiros com k dígitos decimais. Uma forma simples é fazer radix sort com base em cada dígito (começando com o mais signficativo) para separar os números em buckets de 0 a 9.

Exemplo: 45, 24, 39, 58, 23, 32, 25, 47, 18, 54, 51, 42, 43

Ordenar pelo primeiro dígito:

18	24	39	45	58
	23	32	47	54
	25		42	51
			43	

Ordenar pelo segundo dígito:

18	23	32	42	51
	24	39	43	54
	25		45	58
			47	

Algoritmo CGM de Chan e Dehne

Dados n inteiros no intervalo $[0..n^c]$, para contante c.

- **Mistura split sort com radix sort na fase seqüencial.**
- **Não há nenhuma suposição quanto a distribuição dos números.**
- **p processadores com a restrição $n/p \geq p^2$. (Essa restrição pode ser reduzida para $n/p \geq p$.)**
- **Usa 24 rodadas de comunicaão (sendo 6 rodadas com $h = n/p$ e 18 com $h = p$ na h-relação).**
- **Computação local $O(n/p)$.**

Idéia do Algoritmo de Chan e Dehne

Cada processador ordena seus n/p números usando radix sort.

Cada processador seleciona uma *amostra local* de p números: considera os n/p números já ordenados e escolhe um número a cada intervalo de n/p^2.

Cada processador envia a sua *amostra local* de p números ao processador P_0. Note que P_0 vai receber p^2 números. Portanto a memória local deve satisfazer $n/p \geq p^2$.

P_0 ordena todos os p^2 números recebidos usando radix sort.

Idéia do Algoritmo de Chan e Dehne (cont.)

P_0 considera os p^2 números ordenados e seleciona uma *amostra global* de p números pegando um número a cada intervalo de p.

P_0 envia a *amostra global* a todos os processadores.

Cada processador P_i particiona seus n/p números em p buckets $B_{i,0}, B_{i,1}, \ldots, B_{i,p-1}$. $B_{i,j}$ contém valores entre $j-1$-ésima e j-ésima amostra global.

Cada processador P_i envia bucket $B_{i,j}$ ao P_j, para todo j.

Algoritmo CGM de Chan e Dehne

Input: **n inteiros no intervalo de $1, \ldots, n^c$, para a constante fixa c, armazenados em p processadores BSP/CGM, $\frac{n}{p}$ inteiros por processador. $\frac{n}{p} \geq p^2$.**

Output: **Os inteiros ordenados.**

1: **Cada processador ordena localmente seus $\frac{n}{p}$ inteiros, usando o *radix sort*.**

2: **Cada processador seleciona de seus inteiros localmente ordenados, uma amostra (*sample*) de p inteiros com ranks $i(\frac{n}{p^2}), 0 \leq i \leq p-1$. Denominaremos estes inteiros selecionados como amostras locais. Todas as amostras locais são enviadas ao processador P_1. (Note que P_1 receberá um total de p^2 inteiros. Para isso, necessitamos que $O(\frac{n}{p}) \geq p^2$)**

3: **P_1 ordena as p^2 amostras locais recebidas no Passo 2 (usando radix sort) e seleciona uma amostra de p inteiros com ranks $ip, 0 \leq i \leq p-1$. Chamaremos estes inteiros selecionados de amostras globais. As p amostras globais são enviadas *broadcast* a todos os processadores.**

4: **Baseado nas amostras globais recebidas, cada processador P_i, particiona seus $\frac{n}{p}$ inteiros em buckets $B_{i,1}, ...B_{i,p}$ onde $B_{i,j}$ são os inteiros locais com valores entre o $(j-1)$ e j-ésimo elemento das amostras globais.**

5: **Em uma (combinada) h-relação, cada processador $P_i, 1 \leq i \leq p$, envia $B_{i,j}$ para o processador $P_j, 1 \leq j \leq p$. Seja R_j o conjunto de inteiros recebido pelo processador $P_j, 1 \leq j \leq p$, e seja $r_i = |R_i|$.**

6: **Cada processador $P_i, 1 \leq i \leq p$, ordena localmente R_i usando radix sort.**

7: **Uma operação de balanceamento (*balancing shift*) que distribui igualmente todos os inteiros entre os processadores sem alterar sua ordem é realizada como segue: Cada processador $P_i, 1 \leq i \leq p$, envia r_i para P_1. O processador P_1 calcula para cada P_j um vetor A_j de p números indicando quantos de seus inteiros devem ser movidos para o respectivo processadores. Em uma h-relação, todo A_j é enviado a $P_j, 1 \leq j \leq p$. O balanceamento é então executado em uma h-relação subseqüente de acordo com os valores de A_j.**