PRL03 - MNG

Model 2019

Paralelní a distribuované algoritmy

Část 3,4

Řazení

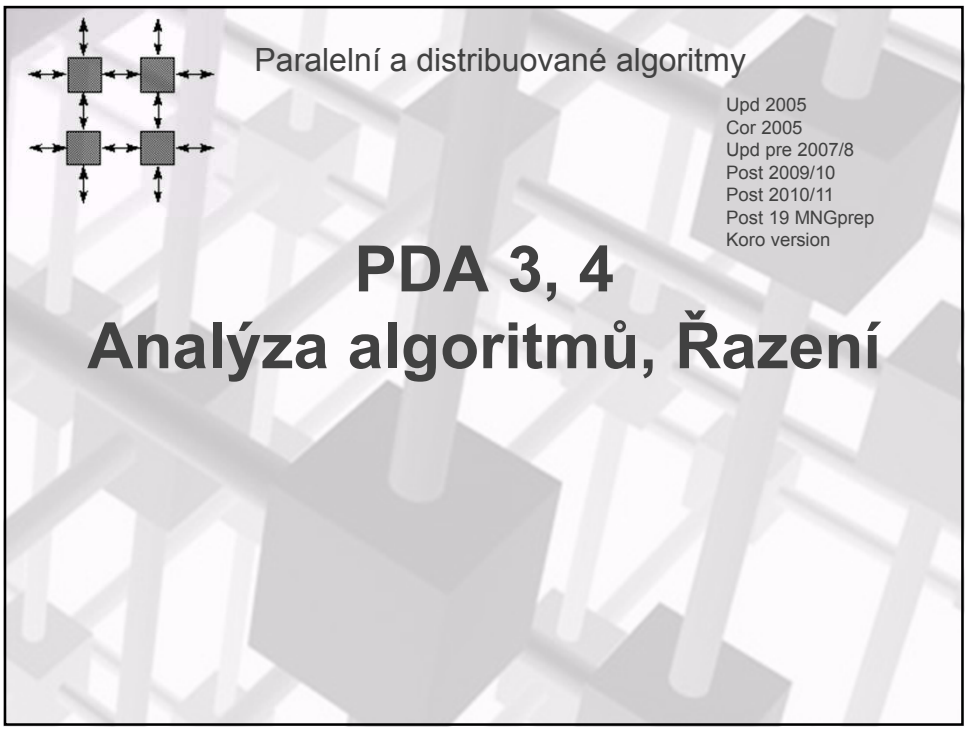
Post 19/20

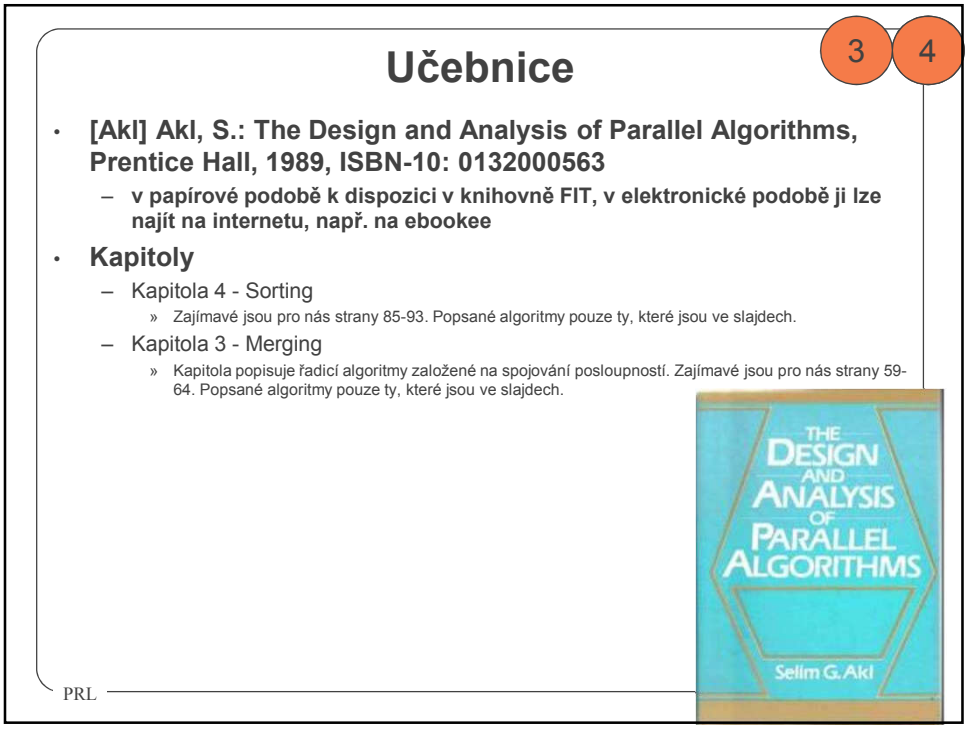
Souhrnné materiály

Ver 0.1

©Petr Hanáček

Paralelní a distribuované algoritmy



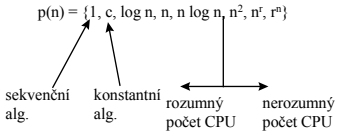


© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

3. ANALÝZA ALGORITMŮ

- **Počet procesorů**
  - (p) potřebných k řešení úlohy v závislosti na velikosti instance n



- **Čas** řešení potřebný k řešení úlohy v jednotkách (krocích) t(n)
- **Cena** paralelního řešení: c(n) = p(n) . t(n)
  - Algoritmus s optimální cenou: $c(n)_{optim} = t_{seq}(n)$
- **Zrychlení** x **Efektivnost**
  - Zrychlení $t_{seq}(n) / t(n)$
  - Efektivnost $t_{seq}(n) / c(n)$
    - <1 neoptimální (přidá se režie)
    - » =1 optimální
    - » >1 ?

PRL 3

4. ŘAZENÍ

- Máme posloupnost $X = \{x_1, ... ,x_n\}$ s n prvky a lineární uspořádání >
- Cílem je vytvořit z prvků x_i novou posloupnost $Y = \{y_1, ..., y_n\}$, kde platí $y_i < y_{i+1}$, i = 1, ... N-1
- V X nejsou žádné dva prvky rovny
- Optimální sekvenční algoritmus - *platí pro řadící algoritmy založené na porovnávání prvku*
  - p(n) = 1
  - t(n) = O(n.log n)
  - c(n) = O(n.log n)

PRL 4

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

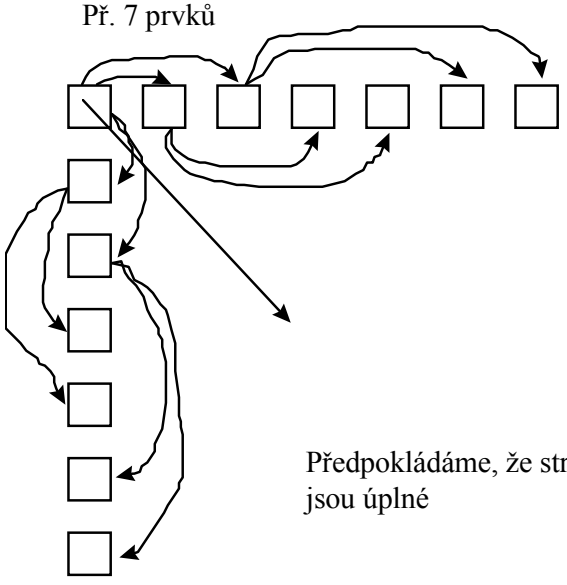
Paralelní a distribuované algoritmy

4.1 Enumeration sort

- Princip: správná pozice každého prvku ve výstupní seřazené posloupnosti je dána počtem prvků, které jsou menší než tento prvek
- Topologie:
  - n^2 procesorů je uspořádáno do mřížky n x n
  - Procesory v každém řádku i jsou propojeny do binárního stromu, kde P(i, j) je propojen s P(i, 2j) a P(i, 2j + 1)
  - Procesory v každém sloupci j jsou propojeny do binárního stromu, kde P(i, j) je propojen s P(2i, j) a P(2i + 1, j)
  - 10 prvků $\Rightarrow$ 100 procesorů
- Ideální algoritmus pro paralelní zpracování
- Vlastnosti: každý procesor
  - Může uložit dva prvky do svých registrů A a B
  - Může porovnat A a B a uložit výsledek do registru RANK
  - Pomocí stromového propojení může předat obsah kteréhokoli registru jinému procesoru
  - Může přičítat k registru RANK

PRL 5

Př. 7 prvků



Předpokládáme, že stromy jsou úplné

PRL 6

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Algoritmus

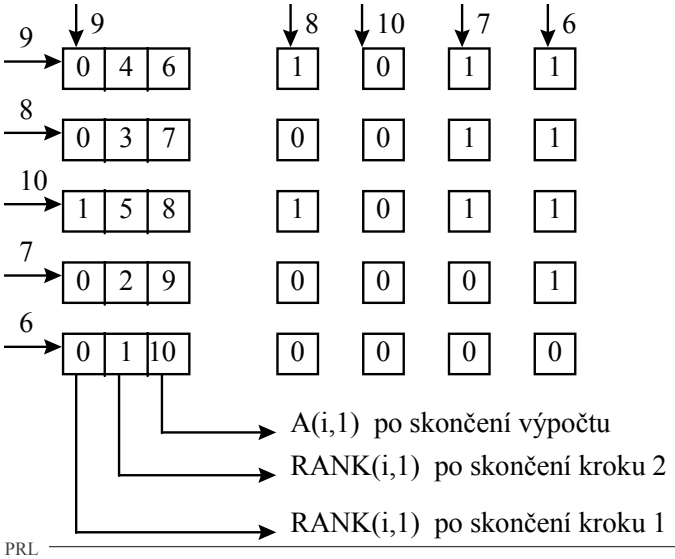
- 1) Každý prvek je porovnán se všemi ostatními pomocí jedné řady procesorů
- 2) Správná pozice prvku je RANK(x_i) = 1 + počet menších prvků
- 3) Každý prvek je zadán na správné místo

```
1) for i=1 to n do in parallel
        1.1) každý procesor P(i, j) v řadě i získá x_i a x_j
                (i, j = 1...n) a uloží je do A(i, j) a B(i, j)

        1.2) if B(i, j) < A(i, j) then RANK(i, j) = 1
                                  else RANK(i, j) = 0
             endif
   endfor
2) for i = 1 to n do in parallel
        2.1) obsah registrů RANK všech procesorů v řadě i je sečten a
          uložen do RANK(i, 1)
        2.2) P(i, 1) spočte RANK(x_i) jako RANK(i, 1) += 1
   endfor
3) for i=1 to n do in parallel
        if  RANK(i, 1)=j then x_i je přesunuto z A(i, j) do A(j, 1)
        endif
   endfor
```

PRL 7

Př. x = {9, 8, 10, 7, 6}, u prvního sloupce pro všechny registry, u ostatních jen RANK



PRL 8

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

- Analýza kroku 1
  - Prvek x_i musí být rozeslán všem procesorům v řadě i a ve sloupci j. Příklad pro řadu i.

```
Procedure PROPAGATE(xi)
(1) A(i, 1) = xi
(2) for k = 1 to ((log n) - 1) do
     for j = 2^(k-1) to 2^k-1 do in parallel        -počet procesorů
            A(i, 2j) = A(i, j)
            A(i, 2j + 1) = A(i, j)
     endfor
    endfor
```

- Tato procedura se složitostí O(log n) je prováděna paralelně pro všechny řady. Podobná procedura se stejnou složitostí slouží pro šíření ve sloupci. Porovnání A a B je v konstantním čase. Složitost kroku 1 je O(log n).

- Analýza kroku 2

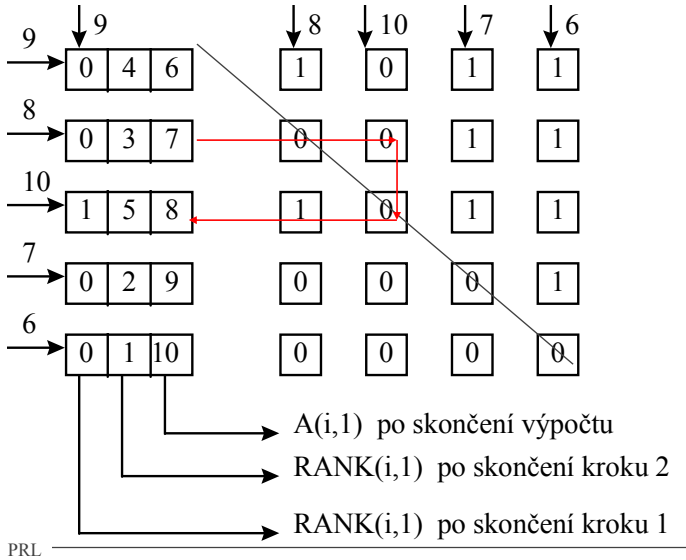
```
Procedure SUM(i)
   for k = ((log n) - 1) downto 1 do
     for j = 2^(k-1) to 2^k-1 do in parallel          - počet procesorů
       RANK(i, j) += RANK(i, 2j) + RANK(i, 2j+1)
                                        sčítání na stromové struktuře
     endfor
   endfor
```

- Což lze provést se složitostí O(log n)
- Analýza kroku 3
  - Procesor P(i, j) zašle x_i do P(RANK(x_i), 1)
    - » 1) P(i, 1) pomocí stromu předá RANK(i, 1) = j do P(i, j)
    - » 2) P(i, i) pomocí stromu ve sloupci i předá A(i, i), t.j. x_i do P(j, i)
    - » 3) P(j, i) předá stromem v řadě j hodnotu x_i do P(j, 1)
  - Každý z těchto kroků má stejnou složitost jako PROPAGATE.
- Krok 3 má stejnou složitost O(log n)

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Př. x = {9, 8, 10, 7, 6}, u prvního sloupce pro všechny registry, u ostatních jen RANK



- Analýza
  - t(n) = O(log n) p(n) = n^2
  - c(n) = O(n^2. log n) což není optimální
- Diskuse
  - Algoritmus je extrémně rychlý O(log n), což znamená zrychlení O(n) krát oproti optimálnímu sekvenčnímu algoritmu.
  - Žádný paralelní algoritmus pro rozumný výpočetní model není rychlejší, bez ohledu na počet procesorů
  - Spotřebovává mnoho procesorů - n^2 je na hranici přijatelnosti
  - Vstupní posloupnost nesmí obsahovat stejné prvky (navrhněte úpravu)

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

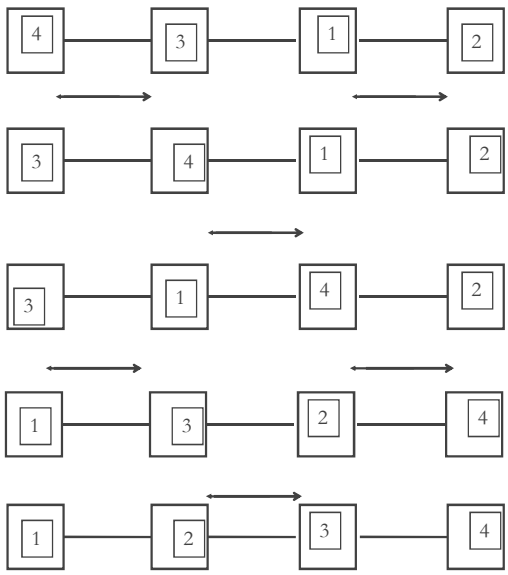
Paralelní a distribuované algoritmy

Odd-even transposition sort

- Lineární pole n procesorů p(n) = n
- Na počátku každý procesor p_i obsahuje jednu z řazených hodnot y_i
- V prvním kroku se každý lichý procesor p_i spojí se svým sousedem p_{i+1} a porovnají své hodnoty je-li $y_i > y_{i+1}$, procesory vymění své hodnoty
- V druhém kroku se každý sudý procesor ...totéž…
- Po n krocích (maximálně) jsou hodnoty seřazeny

```
Algoritmus:
for k = 1 to ⌈n/2⌉ do
  for i = 1, 3, ..., 2.(n/2)-1 do in parallel
    if y_i > y_i+1 then y_i ↔ y_i+1 endif
  endfor
  for i = 2, 4, ..., 2.((n-1)/2) do in parallel
    if y_i > y_i+1 then y_i ↔ y_i+1 endif
  endfor
endfor
```

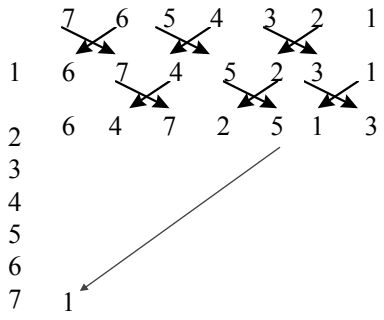
PRL 13



PRL 14

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

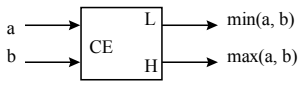


Analýza

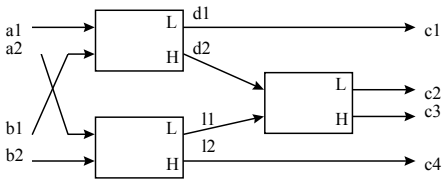
- Každý z kroků (1) a (2) provádí jedno porovnání a dva přenosy - konstantní čas
- Složitost: t(n) = O(n)
- Cena: c(n) = t(n) . p(n) = O(n) . n = $O(n^2)$ *což není optimální*
- Algoritmus má časovou složitost t(n) = O(n), což je to nejlepší, čeho lze při lineární topologii dosáhnout

Odd-even merge sort

- Řadí se speciální sítí procesorů
  - Každý procesor má dva vstupní a dva výstupní kanály
  - Každý procesor umí porovnat hodnoty na svých vstupech, menší dá na výstup L(low), a větší dá výstup H (high)
- Síť 1x1



2x2

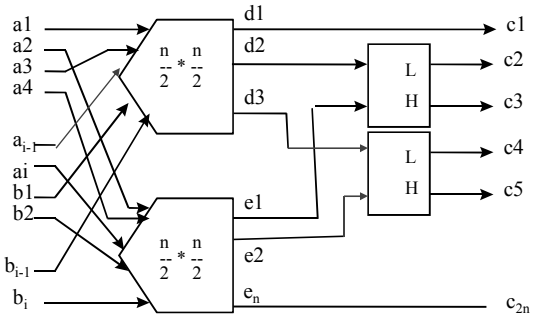


- Seřazené posloupnosti {a1, a2}, {b1, b2} jsou spojeny do seřazené posloupnosti {c1, c2, c3, c4}

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

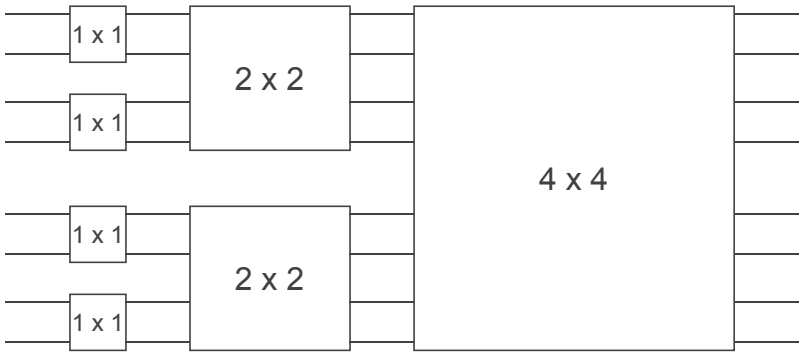
- **Větší sítě n x n**
  - Liché prvky $\{a_1, a_3, ..\}\{b_1, b_3, ...\}$ jsou spojeny sítí n/2 x n/2 do sekvence $\{d_1, d_2, d_3, ...\}$ a podobně sudé prvky do sekvence $\{e_1, e_2,...\}$
  - Finální sekvence $\{c_1, c_2, ...\}$
    - » $c_1 = d_1$
    - » $c_{2i} = \min(d_{i+1}, e_i)$
    - » $c_{2i+1} = \max(d_{i+1}, e_i)$
    - » $c_{2n} = e_n$
- Síť n x n:



PRL 17

Řazení:

- Kaskádou sítí 1x1, 2x2, 4x4, ...



PRL 18

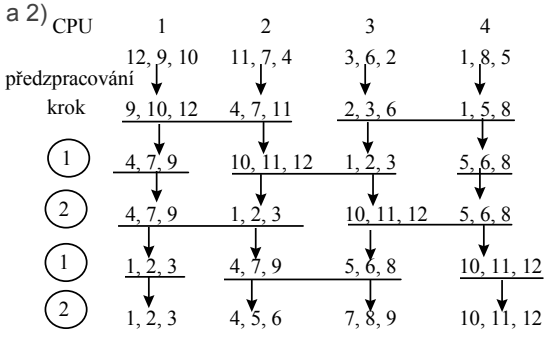
© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

- Analýza
  - Řadíme posloupnost o délce $n=2^m$
  - 1.fáze potřebuje 2^{m-1} CE
  - 2.fáze potřebuje 2^{m-2} sítí 2x2 po 3 procesorech
  - 3.fáze 2^{m-3} sítí 4x4 po 9 procesorech
  - 4.fáze 2^{m-4} sítí po 25 procesorech
  - atd.
- Časová složitost
  - $t(n) = O(m^2) = O(\log^2 n)$
- Cena
  - $c(n) = O(n.\log^4 n)$ což není optimální

Merge-splitting sort

- Lineární pole procesorů p(n) < n
- Je variantou algoritmů lichý-sudý, kde každý procesor obsahuje několik čísel
- Porovnání a výměna je nahrazena operacemi merge-split
- Každý CPU se stará o více prvků
- Každý procesor obsahuje n/p čísel
- Po $\lceil p/2 \rceil$ iteracích (krok 1 a 2) je posloupnost seřazena



© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

```
Algoritmus
for i = 1 to p do in parallel
        procesor Pi seřadí svou posloupnost sekvenčním algoritmem
endfor
for k = 1 to ⌈p/2⌉ do
        1) for i = 1, 3, ..., 2.⌊p/2⌋ do in parallel
                spoj Si a Si+1 do setříděné sekvence Si'
                Si = první polovina Si'
                Si+1 = druhá polovina Si'
           endfor
        2) for = 2, 4, ... , 2.⌊p/2⌋ do in parallel
                ....
           endfor
endfor
```

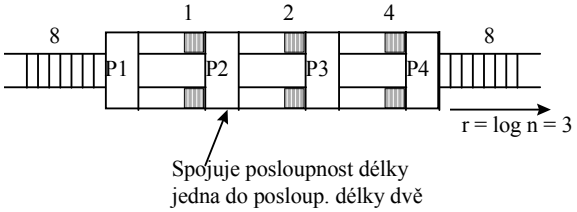
– Analýza
  - » **předzpracování optimálním alg.** **O((n/p)log (n/p))**
  - » **přenos S_{i+1} do P_i** **O(n/p)**
  - » **spojení S_i a S_{i+1} do S_i' optimálním alg.** **2.n/p**
  - » **přenos S_{i+1} do P_{i+1}** **O(n/p)**
  - » **krok 1 nebo 2** **O(n/p)**

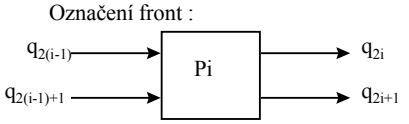
t(n) = O[(n/p) log (n/p)]+O(n) = O((n log n)/p) + O(n)
c(n) = t(n).p = O(n.log n) + O(n.p)
což je optimální pro p ≤ log n

PRL 21

Pipeline Merge sort

- Lineární pole procesorů p(n) = log n + 1
- Data nejsou uložena v procesorech, ale postupně do nich vstupují
- Každý procesor spojuje dvě seřazené posloupnosti délky 2^{i-2}

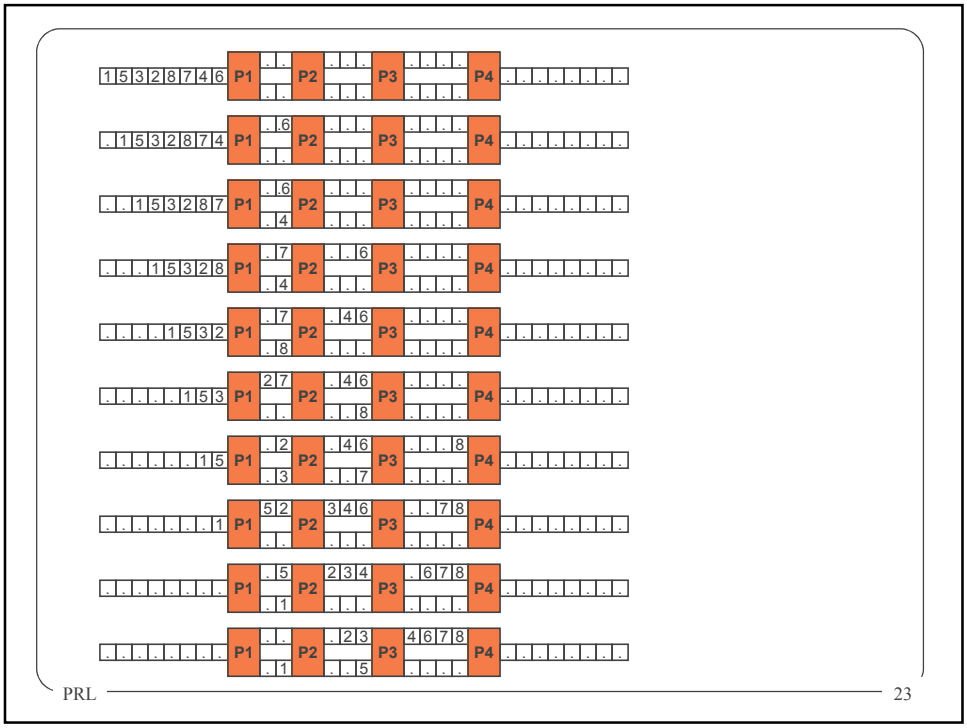




PRL 22

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy



Analýza:

- Procesor P_i začne, když má na jednom vstupu posloupnost délky 2^{i-2} a na druhém 1, tedy začne $2^{i-2}+1$ cyklů po procesoru P_{i-1}.
- P_i tedy začne v cyklu

$$1+\sum_{j=0}^{i-2} 2^j + 1 = 2^{i-1} + i - 1$$

a skončí v cyklu $(n-1) + 2^{i-1} + i - 1$

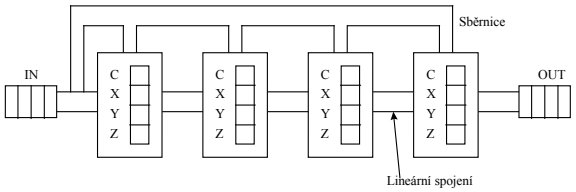
- Algoritmus skončí za:
  - $n + 2^r + r - 1 = 2n + \log n - 1$ cyklů, $t(n) = O(n)$
  - $c(n) = t(n).p(n) = O(n).(\log n + 1) = O(n.\log n)$ což je optimální

PRL 24

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Enumeration sort



- Lineární pole n procesorů, doplněných společnou sběrnicí, schopnou přenést v každém kroku jednu hodnotu
- X_i - prvek x_i
- Y_i - postupně prvky $x_1 ... x_n$
- C_i - počet prvků menších než x_i (t.j. kolikrát byl $Y_i \leq X_i$)
- Z_i - seřazený prvek Y_i

PRL 25

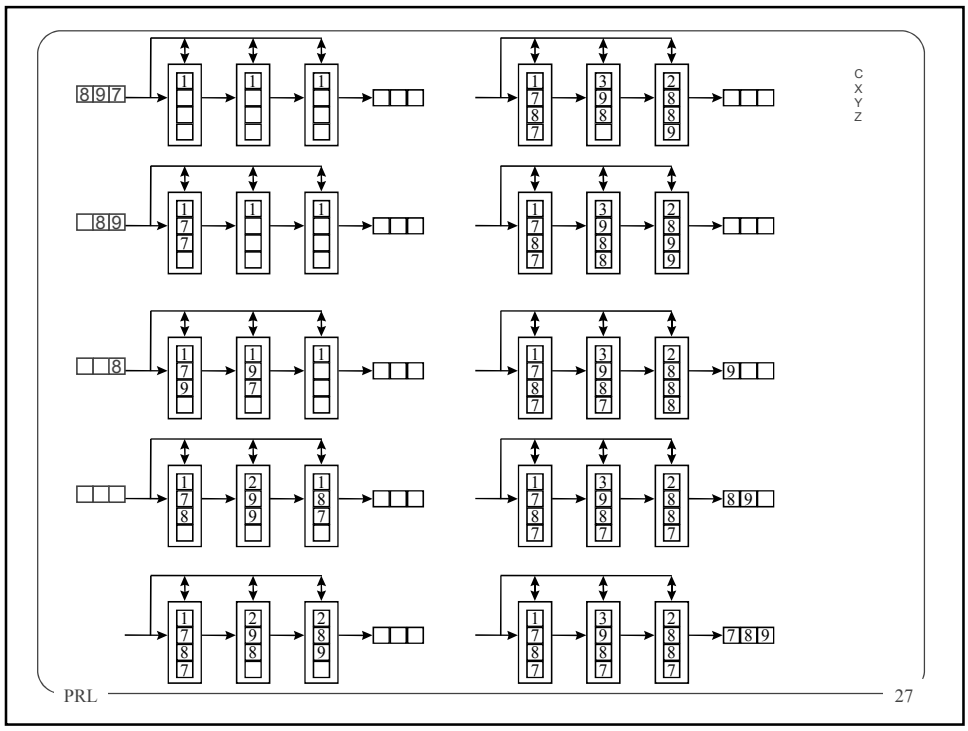
.

- Algoritmus:
  - 1) Všechny registry C se nastaví na hodnotu 1
  - 2) Následující činnosti se opakují 2n krát $1 \leq k \leq 2n$
    - Pokud vstup není vyčerpán, vstupní prvek $\underline{x_i}$ se vloží do X_i (sběrnicí) a do Y_1 (lineárním spojením) a obsah všech registrů Y se posune doprava
    - Každý procesor s neprázdnými registry X a Y je porovná, a je-li X > Y inkrementuje C
    - Je-li k > n (t.j. po vyčerpání vstupu) procesor P_{k-n} pošle sběrnicí obsah svého registru X procesoru P_{Ck-n}, který jej uloží do svého registru Z
  - 3) V následujících $\underline{n}$ cyklech procesory posouvají obsah svých registrů Z doprava a procesor P_n produkuje seřazenou posloupnost

PRL 26

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy




```
Formální algoritmus
1) for i = 1 to n do in parallel
        C_i = 1
   endfor
2) for k = 1 to 2n do
        if k ≤ n then h = 1 else h = k - n endif
        for i = h to n do in parallel
          if (X_i nonempty and Y_i nonempty) and X_i > Y_i then C_i = C_i + 1 endif
        endfor
        for i = h to n - 1 do in parallel
          if Y_i nonempty then Y_i+1 = Y_i endif
        endfor
        if k ≤ n then Y_i = nextinput, X_k = nextinput endif
        if k > n then Z_Ck-n = X_k-n endif
   endfor
3) for k = 1 to n
        output = Z_n
        for i = k to n-1 do in parallel
          Z_i+1 = Z_i
        endfor
   endfor
```

- Analýza
  - Krok 1) je v konstantním čase, krok 2) trvá 2n cyklů, krok 3) trvá n cyklů
  - t(n) = O(n)
  - c(n) = t(n).p(n) = O(n).n = $O(n^2)$, což není optimální
- Poznámky
  - » Výpočet složitosti platí pouze za předpokladu, že přenos hodnoty sběrnicí trvá konstantní dobu bez ohledu na fyzickou vzdálenost procesorů
  - » ~~Algoritmus není schopen řadit vstup obsahující stejné hodnoty *(Navrhněte modifikaci)*~~

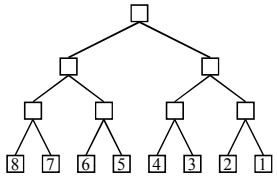
PRL 28

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

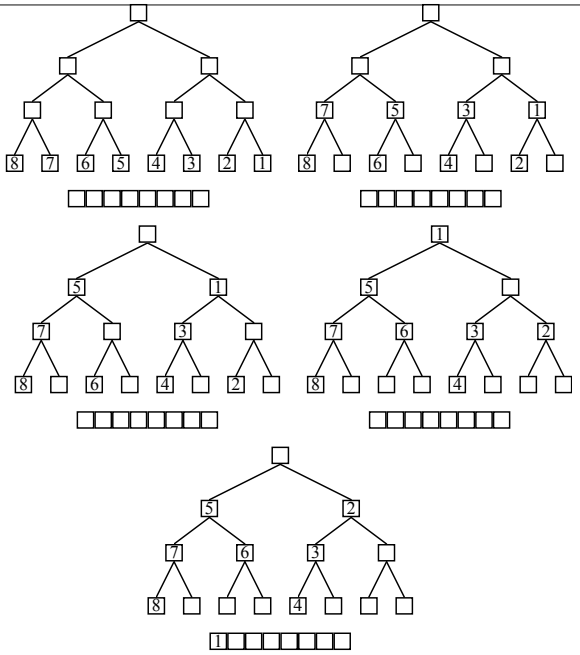
Paralelní a distribuované algoritmy

Minimum Extraction sort

- Strom s n listy, (log n) + 1 úrovněmi a 2n-1 procesory
- Každý listový procesor obsahuje jeden řazený prvek
- Každý nelistový procesor umí porovnat dva prvky
- Algoritmus:
  - Každý list obsahuje jeden prvek
  - Každý nelistový procesor porovná hodnoty svých dvou synů a menší z nich pošle svému otci po (log n) + 1 krocích se minimální prvek dostane do kořenového procesoru
  - Každým dalším krokem se získá další nejmenší prvek



PRL 29



PRL 30

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

- Algoritmus

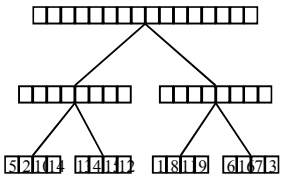
```
1) for all leafs do in parallel
        processor reads one element
    end for
2) for i=1 to 2n+(log n)-1 do
      for all nonleafs do in parallel
        if root and nonempty then output number
        else if nonempty then nothing
             else                                  {i.e. empty nonleaf}
               if  children empty then nothing
               else
                  if one child empty then get number from child
                  else                             {no child empty}
                    get smaller number from both children
                  endif
               endif
             endif
        endif
      endfor
   endfor
```


- Analýza
  - Jelikož strom má (log n)+1 úrovní, první prvek se získá po (log n)+1 krocích. Kořenový procesor potřebuje jeden krok na porovnání a jeden na uložení výsledku do paměti. Každý ze zbylých n-1 prvků spotřebuje 2 kroky.
- $t(n) = 2.n + (\log n) - 1 = O(n)$
- $p(n) = 2.n - 1$
- $c(n) = t(n).p(n) = O(n^2)$ což není optimální

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Bucket sort

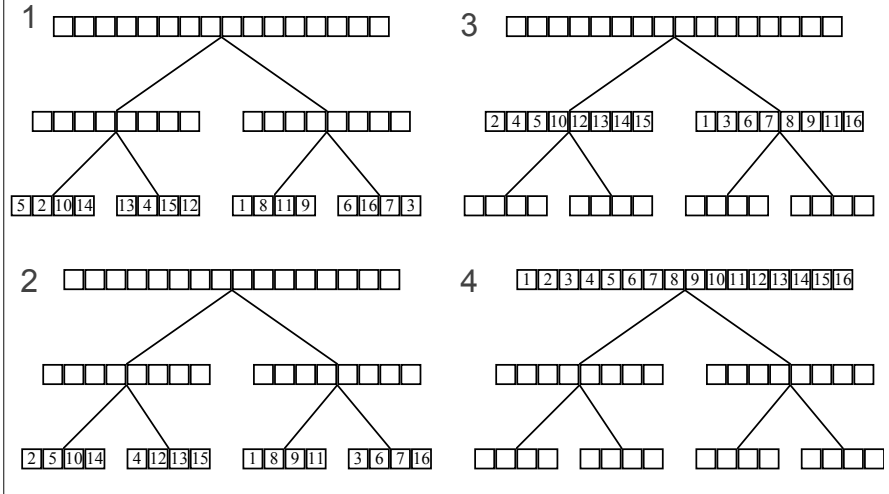


- Řazení stromem procesorů s m listovými procesory $n = 2^m$
- Strom obsahuje 2^{m-1} procesorů, takže $p(n) = (2. \log n) - 1$
- Každý listový procesor obsahuje n/m řazených prvků a umí je seřadit optimálním sekvenčním algoritmem (např. heapsort)
- Každý nelistový procesor umí spojit dvě seřazené posloupnosti optimálním sekvenčním algoritmem
- Algoritmus
  - Řazené prvky se rovnoměrně rozdělí mezi listové procesory
  - Každý list seřadí svou posloupnost

```
for j = 1 to log m do
        for all processors at level (log m)-j do in parallel
                procesor spojí posloupnosti svých synů
        endfor
endfor
```

  - Kořenový procesor uloží výslednou posloupnost do paměti

PRL 33

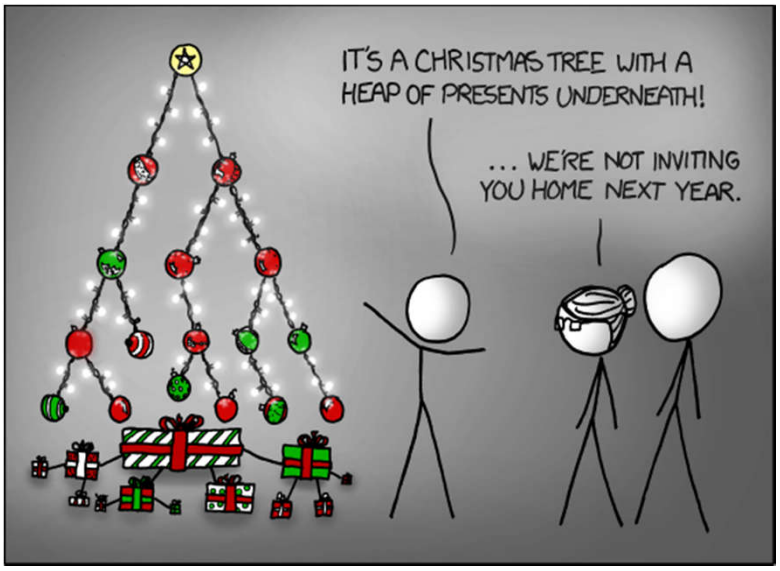


PRL 34

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Tree



PRL 35

- Analýza
  - 1. Každý listový procesor čte n/(log n) prvků, takže složitost je O(n/(log n))
  - 2. Při použití optimálního algoritmu O(r. log r) = O((n/log n)).log(n/(log n)) = O(n)
  - 3. Při j-té iteraci každý procesor na úrovni i = (log m) - j spojí dvě posloupnosti o délce $n/2^i$. Při použití např. straight merge každá j-tá iterace zabere $k.n/2^i$ kroků, kde k je konstantní. Krok 3 tedy trvá

    $$\sum_{i=1}^{(\log m)-1} (k.n)/2^i = O(n)$$

  - 4. O(n)
- Celkově tedy
  - t(n) = O(n) p(n) = O(log n)
  - c(n) = O(n.log n) → optimální

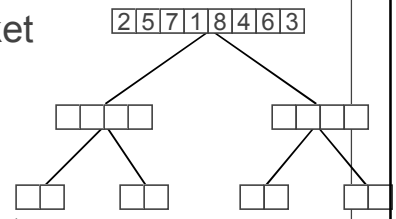
PRL 36

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

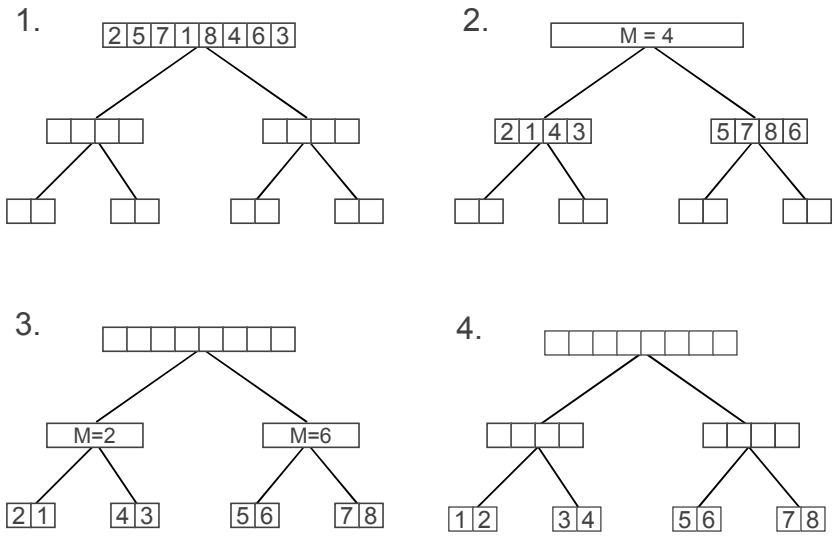
Paralelní a distribuované algoritmy

Median Finding and Splitting

- Stejná architektura jako Bucket Sort
  - Strom procesorů s m listy, $n = 2^m$
  - Procesor na úrovni i zpracovává $n/2^i$ prvků
  - Každý list umí optimální sekvenční sort
- Nový požadavek
  Každý nelistový procesor umí nalézt medián v optimálním čase (např. algoritmus Select s O(n) složitostí)



PRL 37



PRL 38

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

- Algoritmus

```
1.Kořen načte řazenou sekvenci S
2.for i=0 to (log m) - 1 do
        for all processors at level i do in parallel
        2.1) Nalezni ve své sekvenci medián M (sekvenčně)
        2.2) Pro každý prvek x této sekvence
                if x ≤ M then pošli x levému synovi
                        else pošli x pravému synovi
                endif
        endfor
  endfor
3.for all leaf processors do
        3.1) setřiď svou sekvenci sekvenčním algoritmem
        3.2) ulož ji do výstupní vyrovnávací paměti
  endfor
```


- Analýza
  - 1.O(n)
  - 2.Procesor na i-té úrovni hledá medián v posloupnosti délky $n/2^i$, což mu při optimálním algoritmu trvá $O(n/2^i)$. Rozdělení posloupnosti trvá rovněž $O(n/2^i)$, tedy celkem krok 2:

$$\sum_{i=0}^{(\log m)-1} n/2^i = O(n)$$

  - 3.každý procesor řadí posloupnost délky n/log n což mu trvá O(n/(log n))/log(n/(log n))) a výstup uloží v čase O(n/log n)), takže složitost kroku 3 je O(n)

$t(n) = O(n)$ $\quad p(n) = O(\log n)$
$c(n) = O(n.\log n)$ $\rightarrow$ což je optimální

- Diskuse
- Algoritmus se nechová dobře, pokud se v řazené posloupnosti vyskytují stejné prvky

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Řazení na SIMD bez společné paměti

	t(n)	cena optimální?
Speciální topologie		
Enumeration Sort	O(log n)	N
Odd-even Marge Sort	$\log^2 n$	N
Lineární pole procesorů		
Odd-Even Trasposition	n	N
Merge-splitting Sort (agregovaná verze předchozího)		A
Pipeline Merge Sort	n	A
Enumeration Sort	n	N
Mřížka (mesh)		
Mesh Sort	$n^{1/2}$	N
Agregable Mesh Sort		A
Strom		
Minimum Extraction	n	N
Bucket Sort agregovaná verze předchozího		A
Median Finding and Splitting	n	A
Hyperkostka		
Cube Sort $t(n) = O(\log n) .. O(\log^2 n)$ $p(n) = n^2 .. 2n$		

PRL 41

Select (median)

PRL 42

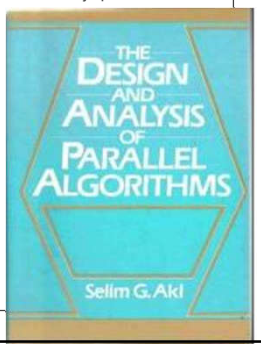
© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Učebnice – Algoritmus Select

3 4

- **Algoritmus select**
- **[Akl] Akl, S.: The Design and Analysis of Parallel Algorithms, Prentice Hall, 1989, ISBN-10: 0132000563**
  - **v papírové podobě k dispozici v knihovně FIT, v elektronické podobě ji lze najít na internetu, např. na ebookee**
- **Kapitoly**
  - Kapitola 2 - Selection
    - » Zajímavá je pro nás prakticky celá kapitola, popisuje jak sekvenční, tak i paralelní verzi algoritmu Select, včetně podrobností, které nejsou ve slajdech a mohou usnadnit pochopení. Zkoušen je pouze v rozsahu slajdů.



PRL

Sequential select

- Hledá k-tý nejmenší prvek v posloupnosti S
- Je-li k = |S| / 2, jde o medián

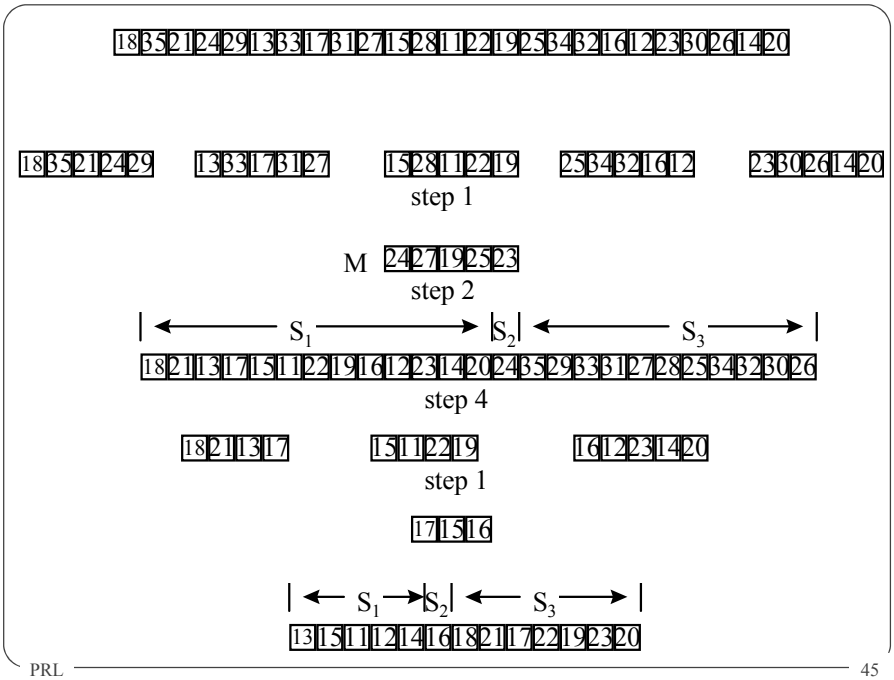
```
Algoritmus
procedure SEQUENTIAL_SELECT(S, k)
(1) if |S| ≤ Q then seřaď S a odpočítej
       else rozděl S na |S|/Q posloupností Si o délce Q prvků
(2) // Seřaď každou posloupnost Si a nalezni její medián M[i]
     for i=1 to |S|/Q do
        M[i] = SEQUENTIAL_SELECT(Si, |Si|/2)
     end for
(3) // Nalezni "medián mediánů" m
    m = SEQUENTIAL_SELECT(M, |M|/2)
(4) L =  {si ∈ S: si < m}
    E =  {si ∈ S: si = m}
    G = { si ∈ S: si > m}
(5) if |L| > k then SEQUENTIAL_SELECT(L,k) // prvek musí být v L
    else if |L| + |E| > k then return m    // prvek musí být v E
    else SEQUENTIAL_SELECT(G, k-|L|-|E|)   // prvek musí být v G
```

- Pro Q>=5 t(n) = O(n)

PRL 44

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy



PRL 45

Parallel select

- Hledá k-tý nejmenší prvek v posloupnosti S
- EREW PRAM s N procesory $P_1 ... P_N$
- Používá sdílené pole M o N prvcích

```
Algoritmus
procedure PARALLEL_SELECT(S, k)
(1) if |S| ≤ 4 then přímo nalezni k-tý prvek
    else rozděl S na N posloupností Si o délce n/N a každou přiřaď
         jednomu procesoru Pi
(2) for i=1 to N do in parallel
       M[i] = SEQUENTIAL_SELECT(Si, |Si|/2)
     end for
(3) m = PARALLEL_SELECT(M, |M|/2)  ← s menším počtem procesorů
(4) L =  {si ∈ S: si < m}
    E =  {si ∈ S: si = m }
    G = { si ∈ S: si > m}
(5) if |L| > k then PARALLEL_SELECT(L,k)
       else if |L| + |E| > k then return m
              else PARALLEL_SELECT(G, k-|L|-|E|)
```

PRL 46

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

- Analýza
  - t(n) = O(n/N) pro n > 4, N < n/log n
  - p(n) = N
  - c(n) = t(n).p(n) = O(n) → což je optimální

Parallel splitting

- Krok 4 algoritmu Parallel select
- Úloha: Je dána posloupnost S a číslo m
  Mají se vytvořit tři posloupnosti:
  $L = \{s_i \in S: s_i < m\}$
  $E = \{s_i \in S: s_i = m\}$
  $G = \{s_i \in S: s_i > m\}$
- Složitost sekvenčního algoritmu je O(n)
- Paralelní řešení - máme N procesorů, které si sekvenci S rozdělí na podposloupnosti S_i o délce n/N

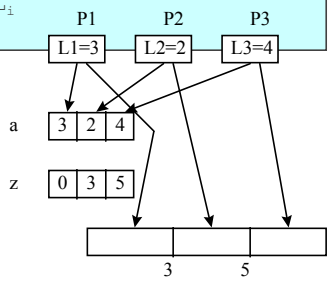
© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Paralelní a distribuované algoritmy

Algoritmus
(i) m se zašle všem procesorům procedurou BROADCAST
(ii) Každý procesor P_i rozdělí svoji sekvenci S_i
na sekvence L_i, E_i, G_i
(iii) Všechny sekvence L_i jsou spojeny do sekvence L
(a stejně tak E_i a G_i) následovně:
Nechť $a_i = |L_i|$
Pro všechny i, kde $1 \leq i \leq N$ se spočte suma:

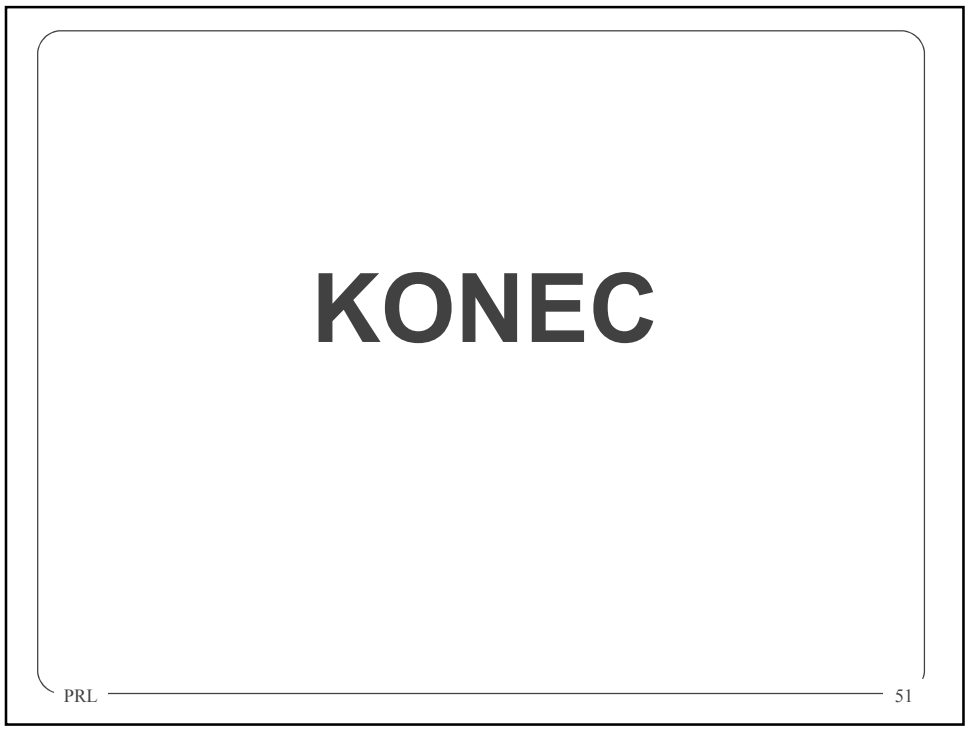
$$z_i = \sum_{j=1}^{i} a_j \qquad \text{a } z_0 = 0$$

(iv) Nyní všechny procesory paralelně
ukládají své posloupnosti
L_i do L tak, že procesor P_i kopíruje L_i
do L od pozice $z_{i-1}+1$.



- Analýza
  - (i) Krok (i) zabere čas O(log N)
  - (ii) Rozdělení se provádí optimálním sekvenčním algoritmem a zabere čas O(n/N)
  - (iii) Hodnoty z_i jsou vlastně suma prefixů a dají se procedurou ALLSUMS spočíst v čase O(log N)
  - (iv) Spojení subsekvencí se provádí paralelně s lineární složitostí a trvá O(n/N)
- Celková časová složitost
  - t(n) = O(log N + n/N) = O(n/N) pro dostatečně malé N
  - Cena c(n) = O(n/N).N = O(n) → což je optimální

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.



© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

4

Sorting

4.1 INTRODUCTION

In the previous two chapters we described parallel algorithms for two comparison problems: selection and merging. We now turn our attention to a third such problem: sorting. Among all computational tasks studied by computer scientists over the past forty years, sorting appears to have received the most attention. Entire books have been devoted to the subject. And although the problem and its many solutions seem to be quite well understood, hardly a month goes by without a new article appearing in a technical journal that describes yet another facet of sorting. There are two reasons for this interest. The problem is important to practitioners, as sorting data is at the heart of many computations. It also has a rich theory: The design and analysis of algorithms is an important area of computer science today thanks mainly to the early work on sorting.

The problem is defined as follows. We are given a sequence $S = \{s_1, s_2, \ldots, s_n\}$ of n items on which a linear order $<$ is defined. The elements of S are initially in random order. The purpose of sorting is to arrange the elements of S into a new sequence $S' = \{s'_1, s'_2, \ldots, s'_n\}$ such that $s'_i < s'_{i+1}$ for $i = 1, 2, \ldots, n-1$. We saw in chapter 1 (example 1.10) that any algorithm for sorting must require $\Omega(n \log n)$ operations in the worst case. As we did in the previous two chapters, we shall assume henceforth, without loss of generality, that the elements of S are numbers (of arbitrary size) to be arranged in nondecreasing order.

Numerous algorithms exist for sorting on a sequential computational model. One such algorithm is given in what follows as the recursive procedure QUICKSORT. The notation $a \leftrightarrow b$ means that the variables a and b exchange their values.

```
procedure QUICKSORT (S)
if |S| = 2 and s2 < s1
then s1 ↔ s2
else if |S| > 2 then
     (1) {Determine m, the median element of S}
                 SEQUENTIAL SELECT (S, ⌈|S|/2⌉)
```

```
    (2) {Split S into two subsequences S1 and S2}
        (2.1) S1 ← {si : si ≤ m} and |S1| = ⌈|S|/2⌉
        (2.2) S2 ← {si : si ≥ m} and |S2| = ⌊|S|/2⌋
    (3) QUICKSORT(S1)
    (4) QUICKSORT(S2)
  end if
end if. □
```

At each level of the recursion, procedure QUICKSORT finds the median of a sequence S and then splits S into two subsequences S_1 and S_2 of elements smaller than or equal to and larger than or equal to the median, respectively. The algorithm is now applied recursively to each of S_1 and S_2. This continues until S consists of either one or two elements, in which case recursion is no longer needed. We also insist that $|S_1| = \lceil |S|/2 \rceil$ and $|S_2| = \lfloor |S|/2 \rfloor$ to ensure that the recursive calls to procedure QUICKSORT are on sequences smaller than S so that the procedure is guaranteed to terminate when all elements of S are equal. This is done by placing all elements of S smaller than m in S_1; if $|S_1| < \lceil |S|/2 \rceil$, then elements equal to m are added to S_1 until $|S_1| = \lceil |S|/2 \rceil$. From chapter 2 we know that procedure SEQUENTIAL SELECT runs in time linear in the size of the input. Similarly, creating S_1 and S_2 requires one pass through S, which is also linear.

For some constant c, we can express the running time of procedure QUICKSORT as

$$\begin{aligned} t(n) &= cn + 2t(n/2) \\ &= O(n \log n), \end{aligned}$$

which is optimal.

Example 4.1

Let $S = \{6, 5, 9, 2, 4, 3, 5, 1, 7, 5, 8\}$. The first call to procedure QUICKSORT produces 5 as the median element of S, and hence $S_1 = \{2, 4, 3, 1, 5, 5\}$ and $S_2 = \{6, 9, 7, 8, 5\}$. Note that $S_1 = \lceil \frac{11}{2} \rceil = 6$ and $S_2 = \lfloor \frac{11}{2} \rfloor = 5$. A recursive call to QUICKSORT with S_1 as input produces the two subsequences $\{2, 1, 3\}$ and $\{4, 5, 5\}$. The second call with S_2 as input produces $\{6, 5, 7\}$ and $\{9, 8\}$. Further recursive calls complete the sorting of these sequences. □

Because of the importance of sorting, it was natural for researchers to also develop several algorithms for sorting on parallel computers. In this chapter we study a number of such algorithms for various computational models. Note that, in view of the $R(n \log n)$ operations required in the worst case to sort sequentially, no parallel sorting algorithm can have a cost inferior to $O(n \log n)$. When its cost is $O(n \log n)$, a parallel sorting algorithm is of course cost optimal. Similarly, a lower bound on the time required to sort using N processors operating in parallel is $\Omega((n \log n)/N)$ for $N \leq n \log n$.

We begin in section 4.2 by describing a special-purpose parallel architecture for sorting. The architecture is a ***sorting network*** based on the odd–even merging

algorithm studied in chapter 3. In section 4.3 a parallel sorting algorithm is presented for an SIMD computer where the processors are connected to form a linear array. Sections 4.4–4.6 are devoted to the shared-memory SIMD model.

4.2 A NETWORK FOR SORTING

Recall how an (r, s)-merging network was constructed in section 3.2 for merging two sorted sequences. It is rather straightforward to use a collection of merging networks to build a sorting network for the sequence $S = \{s_1, s_2, \ldots, s_n\}$, where n is a power of 2. The idea is the following. In a first stage, a rank of $n/2$ comparators is used to create n/2 sorted sequences each of length 2. In a second stage, pairs of these are now merged into sorted sequences of length 4 using a rank of (2, 2)-merging networks. Again, in a third stage, pairs of sequences of length 4 are merged using (4, 4)-merging networks into sequences of length 8. The process continues until two sequences of length $n/2$ each are merged by an $(n/2, n/2)$-merging network to produce a single sorted sequence of length n. The resulting architecture is known as an odd–even sorting *network* and is illustrated in Fig. 4.1 for $S = \{8, 4, 7, 2, 1, 5, 6, 3\}$. Note that, as in the case of merging, the odd–even sorting network is oblivious of its input.

Analysis. As we did for the merging network, we shall analyze the running time, number of comparators, and cost of the odd–even sorting network. Since the size of the merged sequences doubles after every stage, there are log n stages in all.

(i) Running Time. Denote by $s(2^i)$ the time required in the ith stage to merge two sorted sequences of 2^{i-1} elements each. From section 3.2 we have the recurrence

$$\begin{aligned} s(2) &= 1 && \text{for } i = 1, \\ s(2^i) &= s(2^{i-1}) + 1 && \text{for } i > 1, \end{aligned}$$

whose solution is $s(2^i) = i$. Therefore, the time required by an odd–even sorting network to sort a sequence of length n is

$$t(n) = \sum_{i=1}^{\log n} s(2^i) = O(\log^2 n).$$

Note that this is significantly faster than the (optimal) sequential running time of O(n log n) achieved by procedure QUICKSORT.

(ii) Number of Processors. Denote by $q(2^i)$ the number of comparators required in the ith stage to merge two sorted sequences of 2^{i-1} elements each. From section 3.2 we have the recurrence

$$\begin{aligned} q(2) &= 1 && \text{for } i = 1, \\ q(2^i) &= 2q(2^{i-1}) + 2^{i-1} - 1 && \text{for } i > 1, \end{aligned}$$

whose solution is $q(2^i) = (i - 1)2^{i-1} + 1$. Therefore, the number of comparators

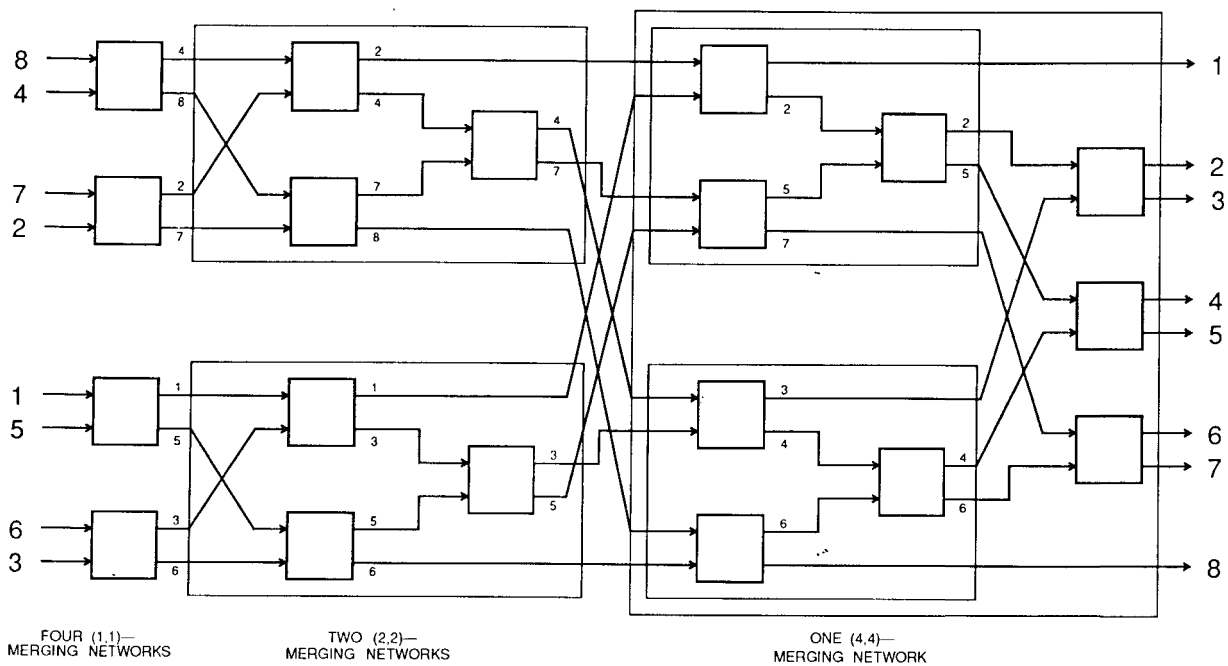


Figure 4.1 Odd–even sorting networks for sequence of eight elements.

needed by an odd–even sorting network to sort a sequence of length n is

$$p(n) = \sum_{i=1}^{\log n} 2^{(\log n)-i} q(2^i)$$
$$= O(n \log^2 n).$$

(iii) ***Cost.*** Since $t(n) = O(\log^2 n)$ and $p(n) = O(n \log^2 n)$, the total number of comparisons performed by an odd–even sorting network, that is, the network's cost, is

$$c(n) = p(n) \times t(n)$$
$$= O(n \log^4 n).$$

Our sorting network is therefore not cost optimal as it performs more operations than the O(n log n) sufficient to sort sequentially.

Since the odd—even sorting network is based on the odd–even merging one, the remarks made in section 3.2 apply here as well. In particular:

(i) The network is extremely fast. It can sort a sequence of length 2^{20} within, on the order of, $(20)^2$ time units. This is to be contrasted with the time required by procedure QUICKSORT, which would be in excess of 20 million time units.

(ii) The number of comparators is too high. Again for $n = 2^{20}$, the network would need on the order of 400 million comparators.

(iii) The architecture is highly irregular and the wires linking the comparators have lengths that vary with n.

We therefore reach the same conclusion as for the merging network of section 3.2: The odd–even sorting network is impractical for large input sequences.

4.3 SORTING ON A LINEAR ARRAY

In this section we describe a parallel sorting algorithm for an SIMD computer where the processors are connected to form a linear array as depicted in Fig. 1.6. The algorithm uses n processors $P_1, P_2, \ldots, P_n$ to sort the sequences $S = \{s_1, s_2, \ldots, s_n\}$. At any time during the execution of the algorithm, processor P_i holds one element of the input sequence; we denote this element by x_i for all $1 \leqslant i \leqslant n$. Initially $x_i = s_i$. It is required that, upon termination, x_i be the ith element of the sorted sequence. The algorithm consists of two steps that are performed repeatedly. In the first step, all odd-numbered processors P_i obtain x_{i+1} from P_{i+1}. If $x_i > x_{i+1}$, then P_i and P_{i+1} exchange the elements they held at the beginning of this step. In the second step, all even-numbered processors perform the same operations as did the odd-numbered ones in the first step. After $\lceil n/2 \rceil$ repetitions of these two steps in this order, no further exchanges of elements can take place. Hence the algorithm terminates with $x_i < x_{i+1}$

for all $1 \leqslant i \leqslant n - 1$. The algorithm is given in what follows as procedure ODD-EVEN TRANSPOSITION.

```
procedure ODD-EVEN TRANSPOSITION (S)
  for j = 1 to ⌈n/2⌉ do
    (1) for i = 1, 3, ..., 2⌊n/2⌋ − 1 do in parallel
          if x_i > x_{i+1}
          then x_i ↔ x_{i+1}
          end if
        end for
    (2) for i = 2, 4, ..., 2⌊(n − 1)/2⌋ do in parallel
          if x_i > x_{i+1}
          then x_i ↔ x_{i+1}
          end if
        end for
  end for. □
```

Example 4.2

Let $S = \{6, 5, 9, 2, 4, 3, 5, 1, 7, 5, 8\}$. The contents of the linear array for this input during the execution of procedure ODD-EVEN TRANSPOSITION are illustrated in Fig. 4.2. Note that although a sorted sequence is produced after four iterations of steps 1 and 2, two more (redundant) iterations are performed, that is, a total of $\lceil \frac{11}{2} \rceil$ as required by the procedure's statement. □

Analysis. Each of steps 1 and 2 consists of one comparison and two routing operations and hence requires constant time. These two steps are executed $\lceil n/2 \rceil$ times. The running time of procedure ODD-EVEN TRANSPOSITION is therefore $t(n) = O(n)$. Since $p(n) = n$, the procedure's cost is given by $c(n) = p(n) \times t(n) = O(n^2)$, which is not optimal.

From this analysis, procedure ODD-EVEN TRANSPOSITION does not appear to be too attractive. Indeed,

(i) with respect to procedure QUICKSORT, it achieves a speedup of $O(\log n)$ only,

(ii) it uses a number of processors equal to the size of the input, which is unreasonable, and

(iii) it is not cost optimal.

The only redeeming feature of procedure ODD-EVEN TRANSPOSITION seems to be its extreme simplicity. We are therefore tempted to salvage its basic idea in order to obtain a new algorithm with optimal cost. There are two obvious ways for doing this: either (1) reduce the running time or (2) reduce the number of processors used. The first approach is hopeless: The running time of procedure ODD-EVEN TRANSPOSITION is the smallest possible achievable on a linear array with n processors. To see this, assume that the largest element in S is initially in P_1 and must therefore move $n - 1$ steps across the linear array before settling in its final position in P,. This requires $O(n)$ time.

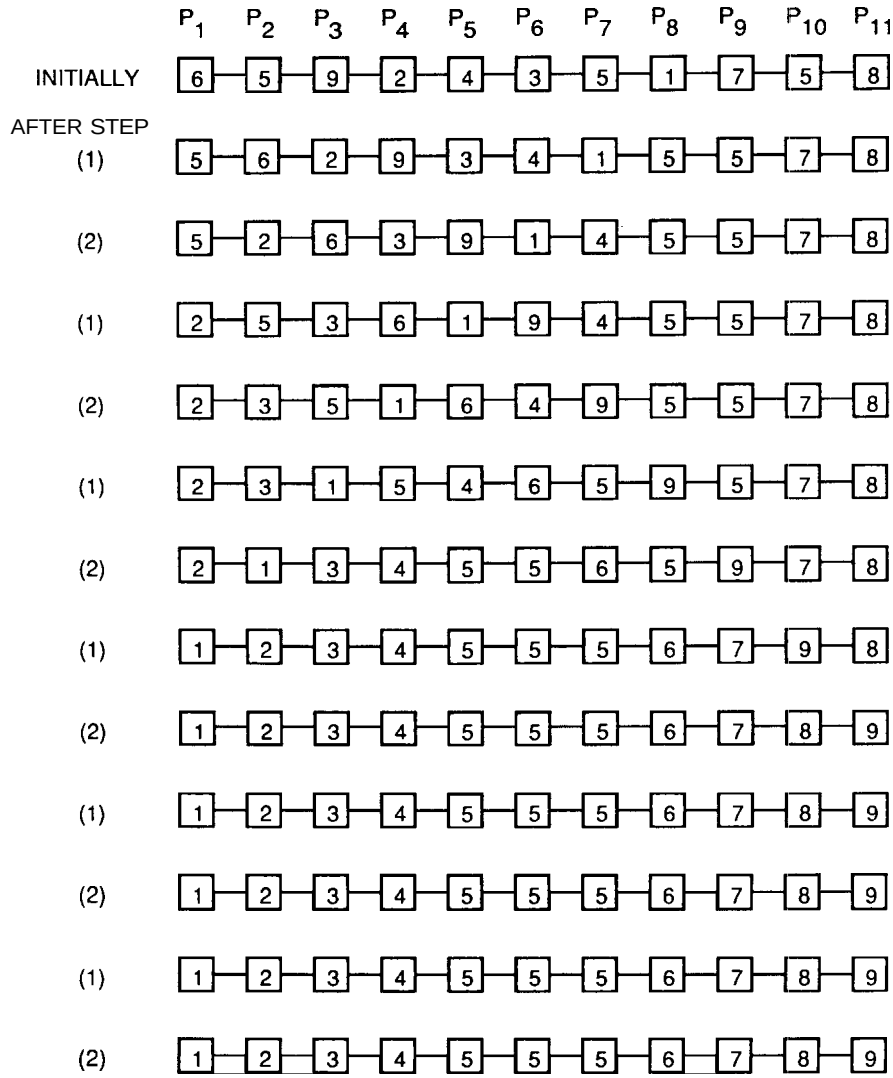


Figure 4.2 Sorting sequence of eleven elements using procedure ODD-EVEN TRANSPOSITION.

Now consider the second approach. If N processors, where $N < n$, are available, then they can simulate the algorithm in $n \times t(n)/N$ time. The cost remains $n \times t(n)$, which as we know is not optimal. A more subtle simulation, however, allows us to achieve cost optimality. Assume that each of the N processors in the linear array holds a subsequence of S of length n/N. (It may be necessary to add some dummy elements to S if n is not a multiple of N.) In the new algorithm, the comparison-exchange

operations of procedure ODD-EVEN TRANSPOSITION are now replaced with merge-split operations on subsequences. Let S_i denote the subsequence held by processor P_i. Initially, the S_i are random subsequences of S. In step *1,* each P_i sorts S_i using procedure QUICKSORT. In step *2.1* each odd-numbered processor P_i merges the two subsequences S_i and S_{i+1} into a sorted sequence $S_i' = \{s_1', s_2', \ldots, s_{2n/N}'\}$. It retains the first half of S_i' and assigns to its neighbor P_{i+1} the second half. Step *2.2* is identical to 2.1 except that it is performed by all even-numbered processors. Steps *2.1* and *2.2* are repeated alternately. After $\lceil N/2 \rceil$ iterations no further exchange of elements can take place between two processors. The algorithm is given in what follows as procedure MERGE SPLIT. When it terminates, the sequence $S = S_1$, $S_2, \ldots, S_N$ is sorted.

procedure MERGE SPLIT *(S)*

Step *1:* **for** i = 1 **to** N **do in parallel**
QUICKSORT (S_i)
end for.

Step *2:* **for** $j = 1$ **to** $\lceil N/2 \rceil$ **do**
(2.1) **for** $i = 1, 3, \ldots, 2\lfloor N/2 \rfloor - 1$ **do in parallel**
(i) SEQUENTIAL MERGE (S_i, S_{i+1}, S_i')
(ii) $S_i \leftarrow \{s_1', s_2', \ldots, s_{n/N}'\}$
(iii) $S_{i+1} \leftarrow \{s'_{(n/N)+1}, s'_{(n/N)+2}, \ldots, s'_{2n/N}\}$
end for
(2.2) **for** $i = 2, 4, \ldots, 2\lfloor (N-1)/2 \rfloor$ **do in parallel**
(i) SEQUENTIAL MERGE (S_i, S_{i+1}, S_i')
(ii) $S_i \leftarrow \{s_1', s_2', \ldots, s_{n/N}'\}$
(iii) $S_{i+1} \leftarrow \{s'_{(n/N)+1}, s'_{(n/N)+2}, \ldots, s'_{2n/N}\}$
end for
end for. □

Example 4.3

Let S = {8, *2, 5, 10, 1, 7, 3, 12, 6, 11, 4, 9)* and N = 4. The contents of the various processors during the execution of procedure MERGE SPLIT for this input is illustrated in *Fig. 4.3.* □

Analysis. Step 1 requires $O((n/N)\log(n/N))$ steps. Transferring S_{i+1} to P_i, merging by SEQUENTIAL MERGE, and returning S_{i+1} to P_{i+1} all require $O(n/N)$ time. The total running time of procedure MERGE SPLIT is therefore

$$\begin{aligned} t(n) &= O((n/N)\log(n/N)) + \lceil N/2 \rceil \times O(n/N) \\ &= O((n \log n)/N) + O(n), \end{aligned}$$

and its cost is

$$c(n) = O(n \log n) + O(nN),$$

which is optimal when $N \leqslant \log n$.

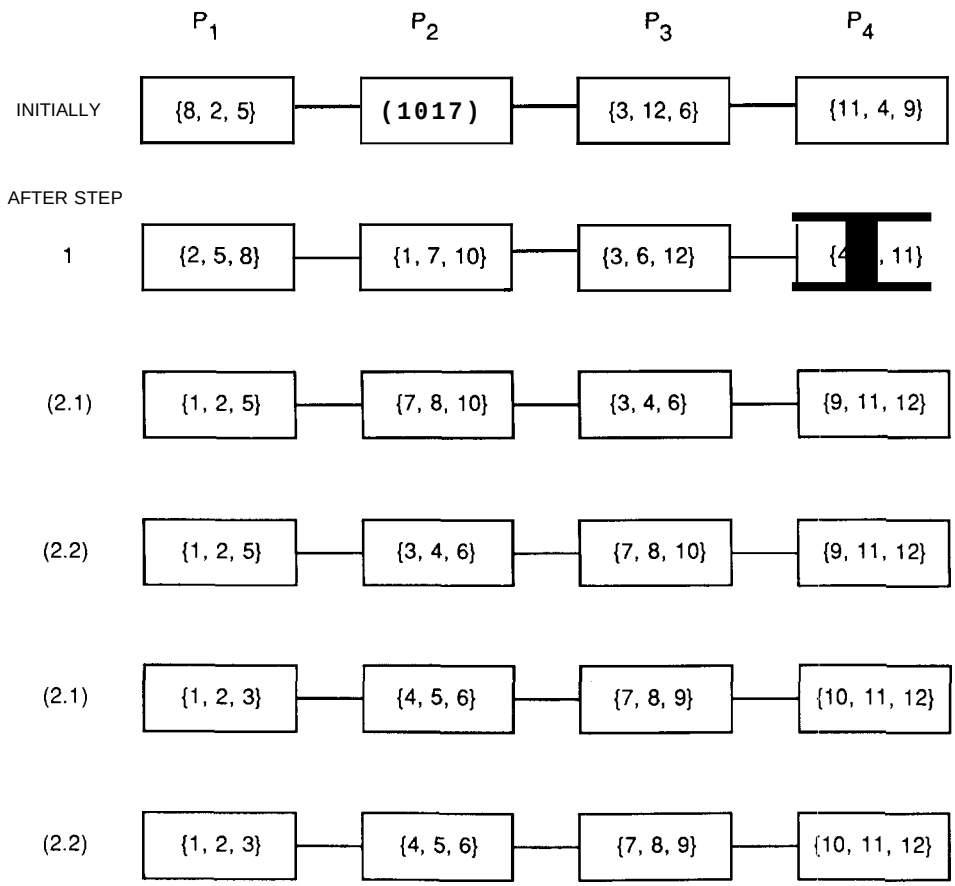


Figure 4.3 Sorting sequence of twelve elements using procedure MERGE SPILIT.

4.4 SORTING ON THE CRCW MODEL

It is time to turn our attention to the shared-memory **SIMD** model. In the present and the next two sections we describe parallel algorithms for sorting on the various incarnations of this model. We begin with the most powerful submodel, the CRCW **SM SIMD** computer. We then proceed to the weaker CREW model (section 4.5), and finally we study algorithms for the weakest shared-memory computer, namely, the EREW model (section 4.6).

Whenever an algorithm is to be designed for the CRCW model of computation, one must specify how write conflicts, that is, multiple attempts to write into the same memory location, can be resolved. For the purposes of the sorting algorithm to be described, we shall assume that write conflicts are created whenever several processors

attempt to write potentially different integers into the same address. The conflict is resolved by storing the ***sum*** of these integers in that address.

Assume that n^2 processors are available on such a CRCW computer to sort the sequence $S = \{s_1, s_2, \ldots, s_n\}$. The sorting algorithm to be used is based on the idea of ***sorting*** by ***enumeration:*** The position of each element s_i of S in the sorted sequence is determined by computing c_i, the number of elements smaller than it. If two elements s_i and s_j are equal, then s_i is taken to be the larger of the two if $i > j$; otherwise s_j is the larger. Once all the c_i have been computed, s_i is placed in position $1 + c_i$ of the sorted sequence. To help visualize the algorithm, we assume that the processors are arranged into n rows of n elements each and are numbered as shown in Fig. 4.4. The shared memory contains two arrays: The input sequence is stored in array S, while the counts c_i are stored in array C. The sorted sequence is returned in array S. The ith row of processors is "in charge" of element s_i: Processors $P(i, 1), P(i, 2), \ldots, P(i, n)$ compute c_i and store s_i in position $1 + c_i$ of S. The algorithm is given as procedure CRCW SORT:

procedure CRCW SORT *(S)*

Step 1: **for** $i = 1$ **to** n **do in parallel**
  for $j = 1$ **to** n **do in parallel**
    if $(s_i > s_j)$ **or** $(s_i = s_j$ **and** $i > j)$
    then $P(i, j)$ writes 1 in c_i
    else $P(i, j)$ writes 0 in c_i
    end if
  end for
end for.

Step 2: **for** $i = 1$ **to** n **do in parallel**
  $P(i, 1)$ stores s_i in position $1 + c_i$ of S
end for. □

Example 4.4

Let $S = \{5, 2, 4, 5\}$. The two elements of S that each of the 16 processors compares and the contents of arrays S and C after each step of procedure CRCW SORT are shown in Fig. 4.5. □

Analysis. Each of steps 1 and 2 consists of an operation requiring constant time. Therefore $t(n) = O(1)$. Since $p(n) = n^2$, the cost of procedure CRCW SORT is

$$c(n) = O(n^2),$$

which is not optimal.

We have managed to sort in constant time on an extremely powerful model that

1. allows concurrent-read operations; that is, each input element s_i is read simultaneously by all processors in row i and all processors in column i;
2. allows concurrent-write operations; that is,
   (i) all processors in a given row are allowed to write simultaneously into the same memory location and

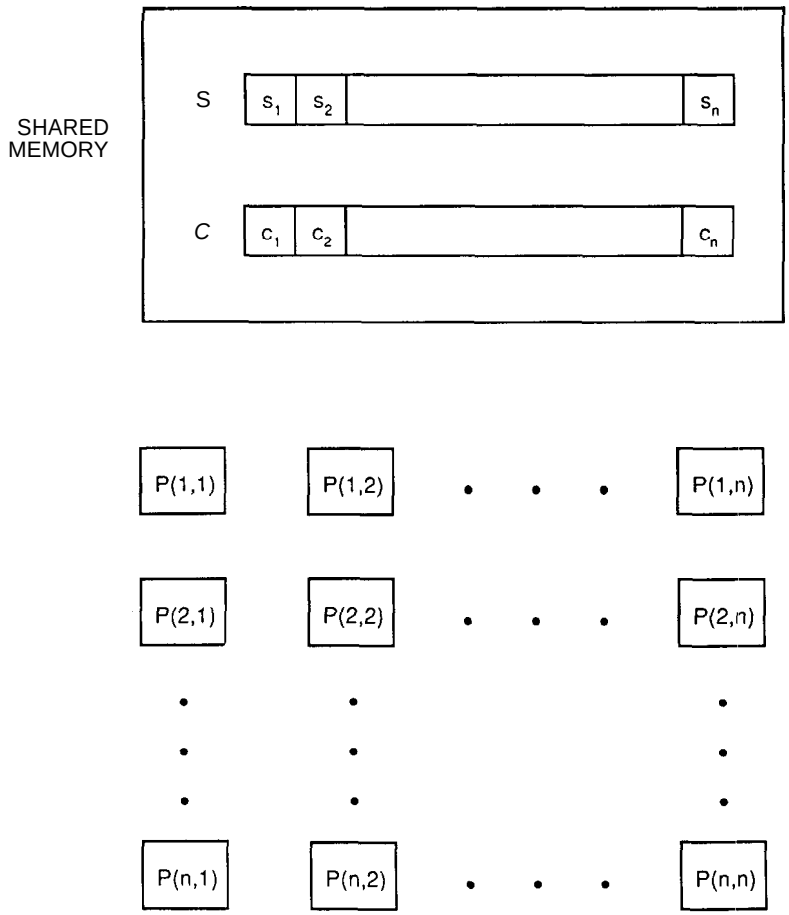


Figure 4.4 Processor and memory organization for sorting on CRCW SM SIMD model.

(ii) the write conflict resolution process is itself very powerful—all numbers to be stored in a memory location are added and stored in constant time; and

3. uses a very large number of processors; that is, the number of processors grows quadratically with the size of the input.

For these reasons, particularly the last one, the algorithm is most likely to be of no great practical value. Nevertheless, procedure CRCW SORT is interesting in its own right: It demonstrates how sorting can be accomplished in constant time on a model that is not only acceptable theoretically, but has also been proposed for a number of contemplated and existing parallel computers.

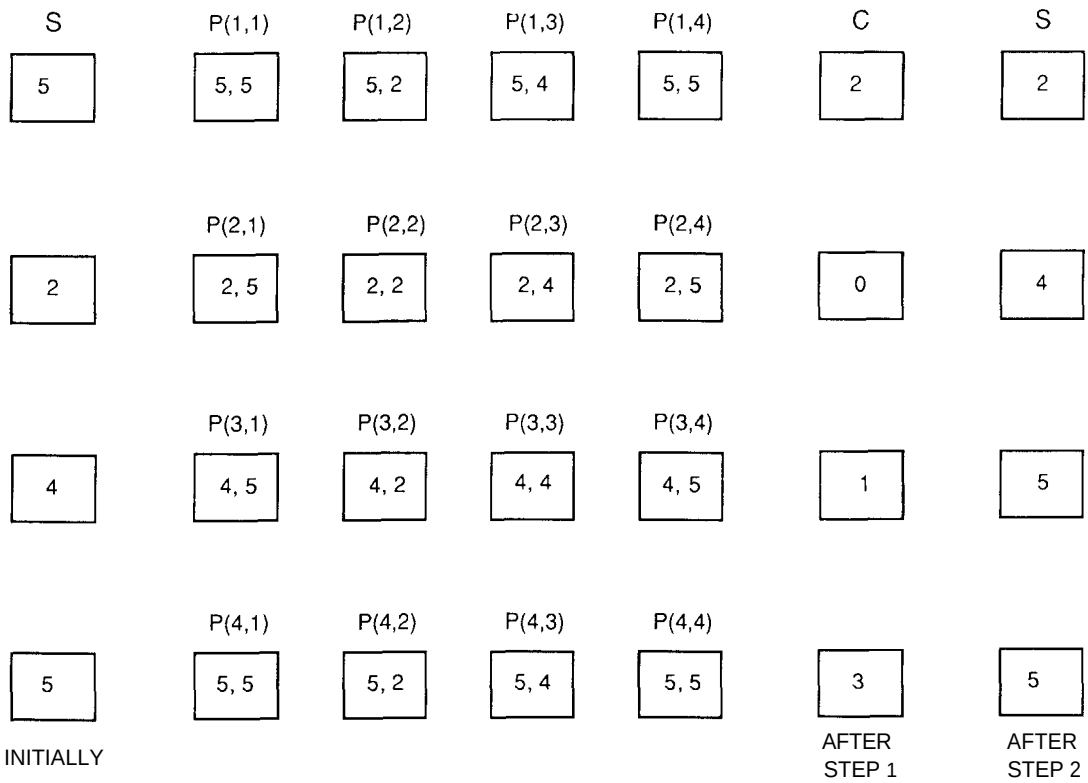


Figure 4.5 Sorting sequence of four elements using procedure CRCW SORT

4.5 SORTING ON THE CREW MODEL

In this section we attempt to deal with two of the objections raised with regards to procedure **CRCW SORT:** its excessive use of processors and its tolerance of write conflicts. Our purpose is to design an algorithm that is free of write conflicts and uses a reasonable number of processors. In addition, we shall require the algorithm to also satisfy our usual desired properties for shared-memory **SIMD** algorithms. Thus the algorithm should have

(i) a sublinear and adaptive number of processors,
(ii) a running time that is small and adaptive, and
(iii) a cost that is optimal.

In sequential computation, a very efficient approach to sorting is based on the idea of merging successively longer sequences of sorted elements. This approach is even more attractive in parallel computation, and we have already invoked it twice in this chapter in sections 4.2 and **4.3.** Once again we shall use a merging algorithm in

order to sort. Procedure CREW MERGE developed in chapter 3 will serve as a basis for the CREW sorting algorithm of this section. The idea is quite simple. Assume that a CREW SM SIMD computer with N processors $P_1, P_2, \ldots, P_N$ is to be used to sort the sequence $S = \{s_1, s_2, \ldots, s_n\}$, where $N \leq n$. We begin by distributing the elements of S evenly among the N processors. Each processor sorts its allocated subsequence sequentially using procedure QUICKSORT. The N sorted subsequences are now merged pairwise, simultaneously, using procedure CREW MERGE for each pair. The resulting subsequences are again merged pairwise and the process continues until one sorted sequence of length n is obtained.

The algorithm is given in what follows as procedure CREW SORT. In it we denote the initial subsequence of S allocated to processor P_i by S_i. Subsequently, S_j^k is used to denote a subsequence obtained by merging two subsequences and P_j^k the set of processors that performed the merge.

procedure CREW SORT *(S)*

Step *1:* **for** i = *1* **to** *N* **do in parallel**
Processor P_i
(1.1) reads a distinct subsequence S_i of S of size n/N
(1.2) QUICKSORT (S_i)
(1.3) $Sf \leftarrow S_i$
(1.4) $Pf \leftarrow \{P_i\}$
end for.

Step 2: *(2.1)* $u \leftarrow 1$
(2.2) $v \leftarrow N$
(2.3) **while** $v > 1$ **do**
(2.3.1) **for** $m = 1$ **to** $\lfloor v/2 \rfloor$ **do in parallel**
(i) $P_m^{u+1} \leftarrow P_{2m-1}^u \cup P_{2m}^u$
(ii) The processors in the set P_m^{u+1} perform
CREW MERGE $(S_{2m-1}^u, S_{2m}^u, S_m^{u+1})$
end for
(2.3.2) **if** v is odd **then** (i) $P_{\lceil v/2 \rceil}^{u+1} \leftarrow P_v^u$
(ii) $S_{\lceil v/2 \rceil}^{u+1} \leftarrow S_v^u$
end if
(2.3.3) $u \leftarrow u + 1$
(2.3.4) $v \leftarrow \lceil v/2 \rceil$
end while. □

Analysis. The dominating operation in step 1 is the call to QUICKSORT, which requires $O((n/N)\log(n/N))$ time. During each iteration of step 2.3, $\lfloor v/2 \rfloor$ pairs of subsequences with $n/\lfloor v/2 \rfloor$ elements per pair are to be merged simultaneously using $N/\lfloor v/2 \rfloor$ processors per pair. Procedure CREW MERGE thus requires $O([(n/\lfloor v/2 \rfloor)/(N/\lfloor v/2 \rfloor)] + \log(n/\lfloor v/2 \rfloor))$, that is, $O((n/N) + \log n)$ time. Since step 2.3 is iterated $\lfloor \log \mathrm{NJ} \rfloor$ times, the total running time of procedure CREW SORT is

$$t(n) = O((n/N)\log(n/N)) + O((n/N)\log N + \log n \log N)$$
$$= O((n/N)\log n + \log^2 n).$$

Since $p(n) = N$, the procedure's cost is given by

$$c(n) = O(n \log n + N \log^2 n),$$

which is optimal for $N \leqslant n/\log n$.

Example 4.5

Let $S = \{2, 8, 5, 10, 15, 1, 12, 6, 14, 3, 11, 7, 9, 4, 13, 16\}$ and $N = 4$. During step 1, processors P_1, P_2, P_3, and P_4 receive the subsequences $S_1 = \{2, 8, 5, 10\}$, $S_2 = \{15, 1, 12, 6\}$, $S_3 = \{14, 3, 11, 7\}$, and $S_4 = \{9, 4, 13, 16\}$, respectively, which they sort locally. At the end of step 1, $S_1^1 = \{2, 5, 8, 10\}$, $S_2^1 = \{1, 6, 12, 15\}$, $S_3^1 = \{3, 7, 11, 14\}$, $S_4^1 = \{4, 9, 13, 16\}$, $P_1^1 = \{P_1\}$, $P_2^1 = \{P_2\}$, $P_3^1 = \{P_3\}$, and $P_4^1 = \{P_4\}$.

During the first iteration of step 2.3, the processors in $P_1^2 = P_1^1 \cup P_2^1 = \{P_1, P_2\}$ cooperate to merge the elements of S_1^1 and S_2^1 to produce $S_1^2 = \{1, 2, 5, 6, 8, 10, 12, 15\}$. Simultaneously, the processors in $P_2^2 = P_3^1 \cup P_4^1 = \{P_3, P_4\}$ merge S_3^1 and S_4^1 into $S_2^2 = \{3, 4, 7, 9, 11, 13, 14, 16\}$.

During the second iteration of step 2.3, the processors in $P_1^3 = P_1^2 \cup P_2^2 = \{P_1, P_2, P_3, P_4\}$ cooperate to merge S_1^2 and S_2^2 into $S_1^3 = \{1, 2, \ldots, 16\}$ and the procedure terminates. □

4.6 SORTING ON THE EREW MODEL

Two of the criticisms expressed with regards to procedure CRCW SORT were addressed by procedure CREW SORT, which adapts to the number of existing processors and disallows multiple-write operations into the same memory location. Still, procedure CREW SORT tolerates multiple-read operations. Our purpose in this section is to deal with this third difficulty. Three parallel algorithms for sorting on the EREW model are described, each representing an improvement over its predecessor. We assume throughout this section that N processors $P_1, P_2, \ldots, P_N$ are available on an EREW SM SIMD computer to sort the sequence $S = \{s_1, s_2, \ldots, s_n\}$, where $N < n$.

4.6.1 Simulating Procedure CREW SORT

The simplest way to remove read conflicts from procedure CREW SORT is to use procedure MULTIPLE BROADCAST. Each attempt to read from memory now takes $O(\log N)$ time. Simulating procedure CREW SORT on the EREW model therefore requires

$$\begin{aligned} t(n) &= O((n/N)\log n + \log n \log N) \times O(\log N) \\ &= O([(n/N) + \log N]\log n \log N) \end{aligned}$$

time and has a cost of

$$c(n) = O((n + N \log N)\log n \log N),$$

which is not cost optimal.

4.6.2 Sorting by Conflict-Free Merging

A more subtle way to avoid concurrent-read operations from the same memory location in procedure CREW SORT is to remove the need for them. This can be accomplished by replacing the call to produce CREW MERGE in step 2.3.1 with a call to procedure EREW MERGE. This step therefore requires $O((n/N) + \log n \log N)$. Since there are $O(\log N)$ iterations of this step, the overall running time of the modified procedure, including step 1, is

$$t(n) = O((n/N)\log(n/N)) + O((n/N)\log N + \log n \log^2 N)$$
$$= O([(n/N) + \log^2 n]\log n),$$

yielding a cost of

$$c(n) = O((n + N \log^2 n)\log n).$$

Therefore the modified procedure is cost optimal when $N \leqslant n/\log^2 n$. This range of optimality is therefore narrower than the one enjoyed by procedure CREW SORT.

4.6.3 Sorting by Selection

Our analysis so far indicates that perhaps another approach should be used if the performance of procedure CREW SORT is to be matched on the EREW model. We now study one such approach. The idea is to adapt the sequential procedure QUICKSORT to run on a parallel computer. We begin by noting that, since $N < n$, we can write $N = n^{1-x}$, where $0 < x < 1$.

Now, let m_i be defined as the $\lceil i(n/2^{1/x})\rceil$th smallest element of S, for $1 \leqslant i \leqslant 2^{1/x} - 1$. The m_i's can be used to divide S into $2^{1/x}$ subsequences of size $n/2^{1/x}$ each. These subsequences, denoted by $S_1, S_2, \ldots, S_j, S_{j+1}, S_{j+2}, \ldots, S_{2j}$, where $j = 2^{(1/x)-1}$, satisfy the following property: Every element of S_i is smaller than or equal to every element of S_{i+1} for $1 \leqslant i \leqslant 2j - 1$. This is illustrated in Fig. 4.6. The subdivision process can now be applied recursively to each of the subsequences S_i until the entire sequence S is sorted in nondecreasing order.

This algorithm can be performed in parallel by first invoking procedure PARALLEL SELECT to determine the elements m_i and then creating the subsequences S_i. The algorithm is applied in parallel to the subsequences $S_1, S_2, \ldots, S_j$ using N/j processors per subsequence. The same is then done with the subsequences $S_{j+1}, S_{j+2}, \ldots, S_{2j}$. Note that the number of processors used to sort each subsequence of size $n/2^{1/x}$, namely, $n^{1-x}/2^{(1/x)-1}$, is exactly the number required for a proper recursive application of the algorithm, that is, $(n/2^{1/x})^{1-x}$.

It is important, of course, that $2^{1/x}$ be an integer of finite size: This ensures that a bound can be placed on the running time and that all the m_i exist. Initially, the N available processors compute x from $N = n^{1-x}$. If x does not satisfy the conditions (i) $\lceil 1/x \rceil \leqslant 10$ (say) and (ii) $n \geqslant 2^{\lceil 1/x \rceil}$, then the smallest real number larger than x an

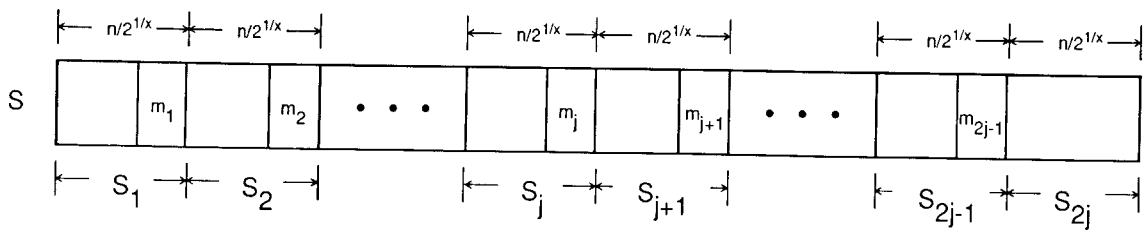


Figure 4.6 Dividing sequence for sorting by selection.

satisfying (i) and (ii) is taken as **x.** Let $k = 2^{\lfloor 1/x \rfloor}$. The algorithm is given as procedure EREW SORT:

```
procedure EREW SORT (S)
  if |S| ≤ k
  then QUICKSORT (S)
  else (1) for i = 1 to k − 1 do
                PARALLEL SELECT (S, ⌈i|S|/k⌉)   {Obtain m_i}
           end for
       (2) S_1 ← {s ∈ S: s ≤ m_1}
       (3) for i = 2 to k − 1 do
                S_i ← {s ∈ S: m_{i−1} ≤ s ≤ m_i}
           end for
       (4) S_k ← {s ∈ S: s ≥ m_{k−1}}
       (5) for i = 1 to k/2 do in parallel
                EREW SORT (S_i)
           end for
       (6) for i = (k/2) + 1 to k do in parallel
                EREW SORT (S_i)
           end for
  end if. □
```

Note that in steps 2–4 the sequence S_i is created using the method outlined in chapter 2 in connection with procedure PARALLEL SELECT. Also in step 3, the elements of S smaller than m_i and larger than or equal to m_{i-1} are first placed in S_i. If $|S_i| < \lceil |S|/k \rceil$, then elements equal to m_i are added to S_i so that either $|S_i| = \lceil |S|/k \rceil$ or no element is left to add to S_i. This is reminiscent of what we did with QUICKSORT. Steps 2 and 4 are executed in a similar manner.

Example 4.6

Let $S = \{5, 9, 12, 16, 18, 2, 10, 13, 17, 4, 7, 18, 18, 11, 3, 17, 20, 19, 14, 8, 5, 17, 1, 11, 15, 10, 6\}$ (i.e., $n = 27$) and let five processors P_1, P_2, P_3, P_4, P_5 be available on an EREW SM SIMD computer (i.e., N = 5). Thus $5 = (27)^{1-x}$, $x \simeq 0.5$, and $k = 2^{\lfloor 1/x \rfloor} = 4$. The working of procedure EREW SORT for this input is illustrated in Fig. 4.7. During step 1, $m_1 = 6$, $m_2 = 11$, and $m_3 = 17$ are computed. The four subsequences S_1, S_2, S_3, and S_4 are created in steps 2–4 as shown in Fig. 4.7(b). In step 5 the procedure is applied recursively and simultaneously to S_1 and S_2. Note that $|S_1| = |S_2| = 7$, and therefore 7^{1-x} is rounded down to 2 (as suggested in chapter 2). In other words two processors are used to sort each of the subsequences S_1 and S_2 (the fifth processor remaining idle). For S_1, processors P_1 and P_2 compute $m_1 = 2$, $m_2 = 4$, and $m_3 = 5$, and the four subsequences $\{1, 2\}$, $\{3, 4\}$, $\{5, 5\}$, and $\{6\}$ are created each of which is already in sorted order. For S_2, processors P_3 and P_4 compute $m_1 = 8$, $m_2 = 10$, and $m_3 = 11$, and the four subsequence:; $\{7, 8\}$, $\{9, 10\}$, $\{10, 11\}$, and $\{11\}$ are created each of which is already in sorted order. The sequence S at the end of step 5 is illustrated in Fig. 4.7(c). In step 6 the procedure is applied recursively and simultaneously to S_3 and S_4. Again since $|S_3| = 7$ and $|S_4| = 6$, 7^{1-x} and 6^{1-x} are rounded down to 2 and two processors are used to sort each of the two subsequences S_3

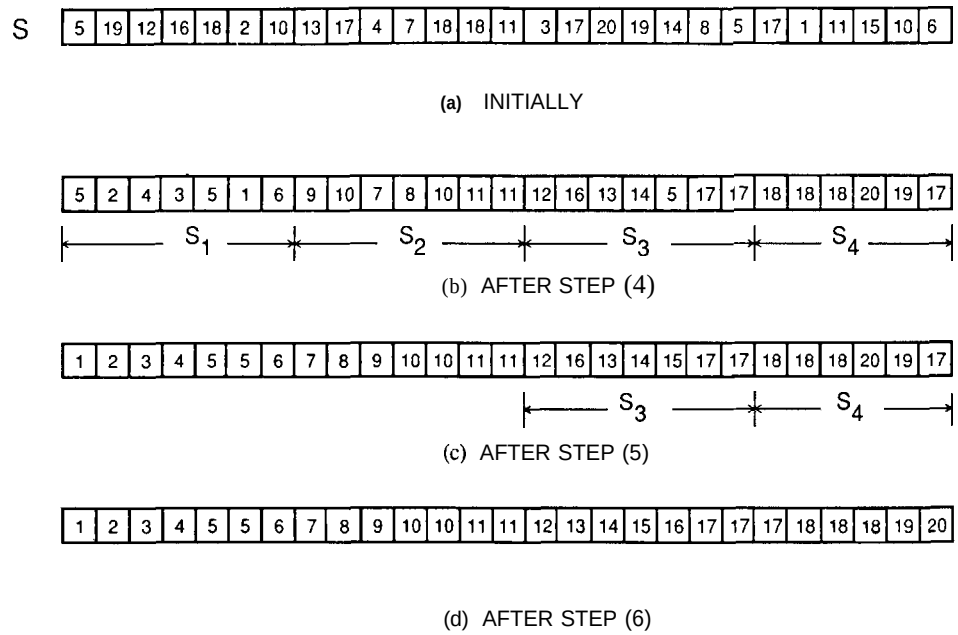


Figure 4.7 Sorting sequence of twenty-seven elements using procedure EREW SORT.

and S_4. For S_3, $m_1 = 13$, $m_2 = 15$, and $m_3 = 17$ are computed, and the four subsequences $\{12, 13\}$, $\{14, 15\}$, $\{16, 17\}$, and $\{17\}$ are created each of which is already sorted. For S_4, $m_1 = 18$, $m_2 = 18$, and $m_3 = 20$ are computed, and the four subsequences $\{17, 18\}$, $\{18, 18\}$, $\{19, 20\}$, and an empty subsequence are created. The sequence S after step 5 is shown in Fig. 4.7(d). □

Analysis. The call to QUICKSORT takes constant time. From the analysis of procedure PARALLEL SELECT in chapter 2 we know that steps 1–4 require cn^x time units for some constant c. The running time of procedure EREW SORT is therefore

$$t(n) = cn^x + 2t(n/k)$$
$$= O(n^x \log n).$$

Since $p(n) = n^{1-x}$, the procedure's cost is given by

$$c(n) = p(n) \times t(n) = O(n \log n),$$

which is optimal. Note, however, that since $n^{1-x} < n/\log n$, cost optimality is restricted to the range $N < n/\log n$.

Procedure EREW SORT therefore matches CREW SORT in performance:

(i) It uses a number of processors N that is sublinear in the size of the input n and adapts to it,

(ii) it has a running time that is small and varies inversely with N, and

(iii) its cost is optimal for $N < n/\log n$.

Procedure EREW SORT has the added advantage, of course, of running on a weaker model of computation that does not allow multiple-read operations from the same memory location.

It is also interesting to observe that procedure EREW SORT is a "mirror image" of procedure CREW SORT in the following way. Both algorithms can be modeled in theory by a binary tree. In procedure CREW SORT, subsequences are input at the leaves, one subsequence per leaf, and sorted locally; they are then merged pairwise by parent nodes until the output is produced at the root. By contrast, in procedure EREW SORT, the sequence to be sorted is input at the root and then split into two independent subsequences $\{S_1, S_2, \ldots, S_j\}$ and $\{S_{j+1}, S_{j+2}, \ldots, S_{2j}\}$; splitting then continues at each node until each leaf receives a subsequence that, once locally sorted, is produced as output.

4.7 PROBLEMS

4.1 Use the (n, n)-merging network defined in problem 3.9 to obtain a network for sorting arbitrary (i.e., not necessarily bitonic) input sequences. Analyze the running time and number of processors used by this network and compare these with the corresponding quantities for the network in section 4.2.

4.2 Consider the following parallel architecture consisting of n^2 processors placed in a square array with n rows and n columns. The processors in each row are interconnected to form a binary tree. The processors in each column are interconnected similarly. The tree interconnections are the only links among the processors. Show that this architecture, known as the mesh of trees, can sort a sequence of n elements in $O(\log n)$ time.

4.3 The odd–even sorting network of section 4.2 uses $O(n \log^2 n)$ processors to sort a sequence of length n in $O(\log^2 n)$ time. For some applications, this may be too slow. On the other hand, the architecture in problem 4.2 sorts in $O(\log n)$ time using n^2 processors. Again, when n is large, this number of processors is prohibitive. Can you design a network that combines the features of these two algorithms, that is, one that uses $O(n \log^2 n)$ processors and sorts in $O(\log n)$ time?

4.4 It may be argued that the number of processors used in problem 4.3, namely, $O(n \log^2 n)$, is still too large. Is it possible to reduce this to $O(n \log n)$ and still achieve an $O(\log n)$ running time?

4.5 Inspect the network obtained in problem 4.1. You will likely notice that it consists of m columns of $n/2$ processors each, where m is a function of n obtained from your analysis. It is required to exploit this regular structure to obtain a sorting network consisting of a single column of $n/2$ processors that sorts a sequence of length n in $O(m)$ time. The idea is to keep the processors busy all the time as follows. The input sequence is fed to the processors and an output is obtained equal to that obtained from the first column of the bitonic sorting network. This output is permuted appropriately and fed back to the processors to obtain the output of the second column. This continues form iterations, until the sequence is fully sorted. Such a scheme is illustrated in Fig. 4.8 for $n = 8$.

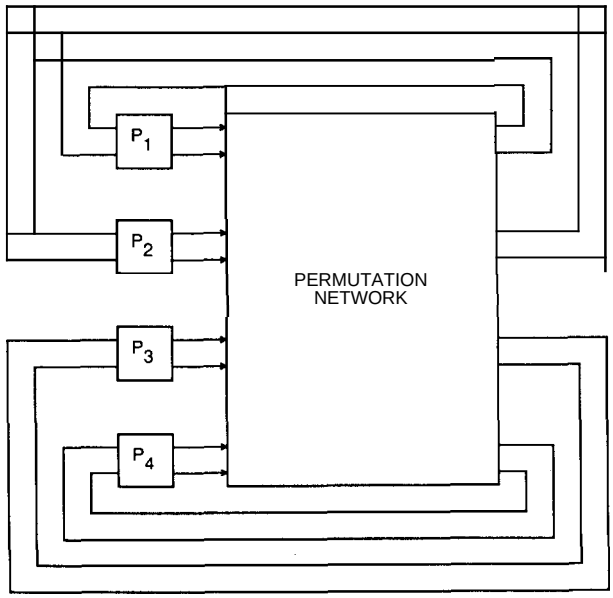


Figure 4.8 Sorting using permutation network.

4.6 The sorting network in problem 4.5 has a cost of $O(nm)$. Is this optimal? The answer, of course, depends on m. If the cost is not optimal, apply the same idea used in procedure MERGE SPLIT to obtain an optimal algorithm.

4.7 Can you design a sorting network that uses $O(n)$ processors to sort a sequence of length n in $O(\log n)$ time?

4.8 Establish the correctness of procedure ODD-EVEN TRANSPOSITION.

4.9 As example **4.2** illustrates, a sequence may be completely sorted several iterations before procedure ODD-EVEN TRANSPOSITION actually terminates. In fact, if the sequence is initially sorted, the $O(n)$ iterations performed by the procedure would be redundant. Is it possible, within the limitations of the linear array model, to modify the procedure so that an early termination is obtained if at any point the sequence becomes sorted?

4.10 Procedure ODD-EVEN TRANSPOSITION assumes that all elements of the input sequence are available and reside initially in the array of processors. It is conceivable that in some applications, the inputs arrive sequentially and are received one at a time by the leftmost processor P_1. Similarly, the output is produced one element at a time from P_1. Modify procedure ODD-EVEN TRANSPOSITION so that it runs under these conditions and completes the sort in exactly the same number of steps as before (i.e., without an extra time penalty for input and output).

4.11 When several sequences are queued for sorting, the procedure in problem 4.9 has a period of 2n. Show that this period can be reduced to n by allowing both P_1 and P_n to handle input and output. In this way, m sequences of n elements each are sorted in $(m + 1)n$ steps instead of 2mn.

4.12 In section **4.3** we showed how procedure ODD-EVEN TRANSPOSITION can be modified so that its cost becomes optimal. Show that it is possible to obtain a cost-optimal sorting algorithm on the linear array for the case of sequential input. One approach to consider is the following. For a sequence of length n, the linear array consists of $1 + \log n$ processors. The leftmost processor receives the input, the rightmost produces the output. Each processor is connected to its neighbors by two lines, as shown in Fig. **4.9** for $n = 8$. This array can be made to sort in $O(n)$ time by implementing an adapted version of the sequential procedure Mergesort. This procedure consists of $\log n$ stages. In stage i sorted subsequences of length 2^i are created, $i = 1, 2, \ldots, \log n$. In the parallel adaptation, the steps are run overlapped on the linear array.

4.13 In procedure MERGE SPLIT each processor needs at least $4n/N$ storage locations to merge two sequences of length n/N each. Modify the procedure to require only $1 + n/N$ locations per processor.

4.14 A variant of the linear array that uses a bus was introduced in problem *2.9.* Design an algorithm for sorting on this model, where P_1 receives the input sequence of size n and P_n produces the output.

4.15 The n elements of a sequence are input to an $n^{1/2} \times n^{1/2}$ mesh-connected SIMD computer, one element per processor. It is required to sort this sequence in row-major order. Derive a lower bound on the running time required to solve this problem.

4.16 Use the results of problems **3.6** and **3.7** to obtain an algorithm for odd–even sorting on an $m \times m$ mesh-connected SIMD computer. Analyze your algorithm.

4.17 Is the algorithm obtained in problem **4.16** cost optimal? If not, apply the same idea used in procedure MERGE SPLIT to obtain a cost-optimal algorithm.

4.18 Use the results of problems 3.12 and **3.13** to obtain an algorithm for bitonic sorting on an $m \times m$ mesh-connected SIMD computer. Analyze your algorithm.

4.19 Repeat problem **4.17** for the algorithm in problem **4.18.**

4.20 The algorithm in problem **4.16** returns a sequence sorted in row-major order. Another indexing that may sometimes be desirable is known as ***snakelike row-major order:*** The ith element resides in row j and column k, where

$$i = \begin{cases} jm + k + 1 & \text{for } j \text{ even,} \\ jm + m - k & \text{for } j \text{ odd.} \end{cases}$$

This is illustrated in Fig. **4.10** for $n = 16$. Show that after a sequence has been sorted into row-major order, its elements may be rearranged into snakelike row-major order in $2(n^{1/2} - 1)$ routing steps.

4.21 Another indexing for sequences sorted on two-dimensional arrays is the ***shuffled row-major order.*** Let element i, $1 \leqslant i \leqslant n$, reside in row j and column k in a row-major ordering. If i' is the integer obtained by applying a perfect shuffle to the bits in the binary representation of $i - 1$, then element $i' + 1$ occupies position (j, k) in a shuffled row-major indexing. This is

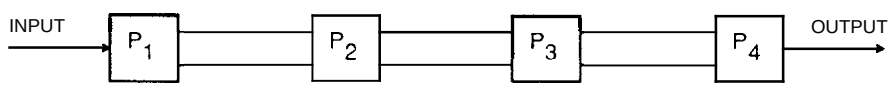


Figure 4.9 Cost-optimal sorting on linear array for case of sequential input.

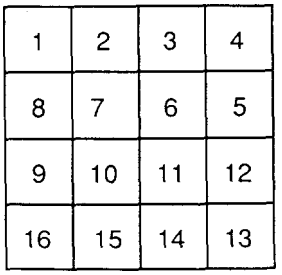


Figure 4.10 Snakelike row-major order.

illustrated in Fig. *4.11* for n = 16. Show that if n elements have already been sorted according to row-major order and if each processor can store $n^{1/2}$ elements, then the n elements can be sorted into shuffled row-major order using an additional $4(n^{1/2} - 1)$ routing steps.

4.22 A variant of the mesh interconnection network that uses a bus was introduced in problem *2.10.* Repeat problem *4.15* for this model.

4.23 Design a parallel algorithm for sorting on the model of problem *2.10.*

4.24 Design an algorithm for sorting on a tree-connected SIMD computer. The input sequence is initially distributed among the leaves of the tree. Analyze the running time, number of processors used, and cost of your algorithm.

4.25 Repeat problem *4.24* for the case where the sequence to be sorted is presented to the root.

4.26 Derive a lower bound for sorting a sequence of length n on the pyramid machine defined in problem 3.16.

4.27 Design an algorithm for sorting on the pyramid machine.

4.28 Show that any parallel algorithm that uses a cube-connected SIMD computer with N processors to sort a sequence of length n, where $N \geqslant n$, requires $\Omega(\log N)$ time.

4.29 Implement the idea of sorting by enumeration on a cube-connected SIMD computer and analyze the running time of your implementation.

4.30 Show that any parallel algorithm that uses the perfect shuffle interconnection network with N processors to sort a sequence of length n, where $N = 2^m \geqslant n$, requires $\Omega(\log N)$ time.

4.31 Consider a CRCW SM SIMD computer where write conflicts are resolved as follows: The write operation is allowed if and only if all processors writing simultaneously in the same memory location are attempting to store the same value. Describe an algorithm for this

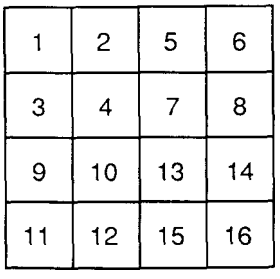


Figure 4.11 Shuffled row-major order.

model that can determine the minimum of n numbers $\{x_1, x_2, \ldots, x_n\}$ in constant time using n^2 processors. If more than one of the numbers qualify, the one with the smallest subscript should be returned.

4.32 Show how procedure CRCW SORT can be modified to run on an EREW model and analyze its running time.

4.33 Show that procedure CREW SORT can be simulated on an EREW computer in $O([(n/N) + \log^2 n]\log n)$ time if a way can be found to distinguish between simple read operations and multiple-read operations, as in problem 3.20.

4.34 In procedure EREW SORT, why are steps 5 and 6 not executed simultaneously?

4.35 Derive an algorithm for sorting by enumeration on the EREW model. The algorithm should use $n^{1+1/k}$ processors, where k is an arbitrary integer, and run in O(k log n) time.

4.36 Let the elements of the sequence S to be sorted belong to the set $\{0, 1, \ldots, m-1\}$. A sorting algorithm known as sorting by bucketing first distributes the elements among a number of buckets that are then sorted individually. Show that sorting can be completed in $O(\log n)$ time on the EREW model using n processors and $O(mn)$ memory locations.

4.37 The amount of memory required for bucketing in problem 4.36 can be reduced when the elements to be sorted are binary strings in the interval $[0, 2^b - 1]$ for some b. The algorithm consists of b iterations. During iteration i, $i = 0, 1, \ldots, b-1$, each element to be sorted is placed in one of two buckets depending on whether its ith bit is 0 or 1; the sequence is then reconstructed using procedure ALLSUMS so that all elements with a 0 ith bit precede all the elements with a 1 ith bit. Show that in this case sorting can be completed in O(b log n) time using $O(n)$ processors and $O(n)$ memory locations.

4.38 Assume that an interconnection network SIMD computer with n processors can sort a sequence of length n in $O(f(n))$ time. Show that this network can simulate an algorithm requiring time T on an EREW SM SIMD computer with n memory locations and n processors in $O(Tf(n))$ time.

4.39 Design an asynchronous algorithm for sorting a sequence of length n by enumeration on a multiprocessor computer with N processors.

4.40 Adapt procedure QUICKSORT to run on the model of problem 4.39.

4.8 BIBLIOGRAPHICAL REMARKS

An extensive treatment of parallel sorting is provided in [Akl 2]. Taxonomies of parallel sorting algorithms can be found in [Bitton] and [Lakshmivarahan]. The odd–even sorting network was first presented in [Batcher]. Other sorting networks are proposed in [Lee:], [Miranker], [Tseng], [Winslow], and [Wong]. The theoretically fastest possible network for sorting using $O(n)$ processors is described in [Leighton] based on ideas appearing in [Ajtai]: It sorts a sequence of length n in $O(\log n)$ time and is therefore cost optimal. However, the asymptotic expression for the running time of this network hides an enormous constant, which makes it infeasible in practice.

Procedure ODD-EVEN TRANSPOSITION is attributed to [Demuth]. The idea on which procedure MERGE SPLIT is based comes from [Baudet]. Other algorithms for sorting on a linear array are described in [Akl 1], [Todd], and [Yasuura]. Parallel sorting algorithms for a variety of interconnection-network SIMD computers have been proposed. These include algorithms for the perfect shuffle ([Stone]), the mesh ([Kumar], [Nassimi 1], and [Thompson]), the tree ([Bentley], [Horowitz 2], and [Orenstein]), the pyramid ([Stout]), and the cube ([Nassimi 2]).

3

Merging

3.1 INTRODUCTION

We mentioned in chapter 2 that selection belongs to a class of problems known as comparison problems. The second such problem to be studied in this book is that of ***merging.*** It is defined as follows: Let A = (a,, a_2, ...,a,) and B = (b,, b_2, ..., b_s) be two sequences of numbers sorted in nondecreasing order; it is required to ***merge*** A and ***B,*** that is, to form a third sequence C = $\{c_1, c_2, \ldots, c_{r+s}\}$, also sorted in nondecreasing order, such that each c_i in C belongs to either A or B and each a_i and each b_i appears exactly once in C. In computer science, merging arises in a variety of contexts including database applications in particular and file management in general. Many of these applications, of course, involve the merging of nonnumeric data. Furthermore, it is often necessary once the merging is complete to delete duplicate entries from the resulting sequence. A typical example is the merging of two mailing lists each sorted alphabetically. These variants offer no new insights and can be handled quite easily once the basic problem stated above has been solved.

Merging is very well understood in the sequential model of computation and a simple algorithm exists for its solution. In the worst case, when $r = s = n$, say, the algorithm runs in $O(n)$ time. This is optimal since every element of A and B must be examined at least once, thus making $\Omega(n)$ steps necessary in order to merge. Our purpose in this chapter is to show how the problem can be solved on a variety of parallel computational models. In view of the lower bound just stated, it should be noted that $\Omega(n/N)$ time is needed by any parallel merging algorithm that uses N processors.

We begin in section 3.2 by describing a special-purpose parallel architecture for merging. A parallel algorithm for the CREW SM SIMD model is presented in section 3.3 that is adaptive and cost optimal. Since the algorithm invokes a sequential procedure for merging, that procedure is also described in section 3.3. It is shown in section 3.4 how the concurrent-read operations can be removed from the parallel algorithm of section 3.3 by simulating it on an EREW computer. Finally,,an adaptive and optimal algorithm for the EREW SM SIMD model is presented in section 3.5 whose running time is smaller than that of the simulation in section 3.4. The algorithm

is based on a sequential procedure for finding the median of two sorted sequences, also described in section 3.5.

3.2 A NETWORK FOR MERGING

In chapter 1 we saw that special-purpose parallel architectures can be obtained in any one of the following ways:

(i) using specialized processors together with a conventional interconnection network,

(ii) using a custom-designed interconnection network to link standard processors, or

(iii) using a combination of (i) and (ii).

In this section we shall take the third of these approaches. Merging will be accomplished by a collection of very simple processors communicating through a special-purpose network. This special-purpose parallel architecture is known as an (r, s)-*merging* network. All the processors to be used are identical and are called comparators. As illustrated by Fig. 3.1, a comparator receives two inputs and produces two outputs. The only operation a comparator is capable of performing is to compare the values of its two inputs and then place the smaller and larger of the two on its top and bottom output lines, respectively.

Using these comparators, we proceed to build a network that takes as input the two sorted sequences $A = \{a_1, a_2, \ldots, a_r\}$ and $B = \{b_1, b_2, \ldots, b_s\}$ and produces as output a single sorted sequence $C = \{c_1, c_2, \ldots, c_{r+s}\}$. The following presentation is greatly simplified by making two assumptions:

1. the two input sequences are of the same size, that is, $r = s = n \geq 1$, and
2. n is a power of 2.

We begin by considering merging networks for the first three values of n. When $n = 1$, a single comparator clearly suffices: It produces as output its two inputs in

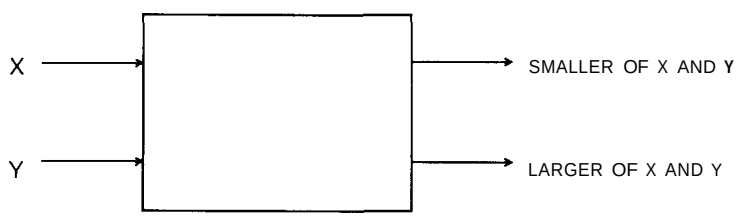


Figure 3.1 Comparator.

sorted order. When $n = 2$, the two sequences $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ are correctly merged by the network in Fig. 3.2. This is easily verified. Processor P_1 compares the smallest element of A to the smallest element of B. Its top output must be the smallest element in C, that is, c_1. Similarly, the bottom output of P_2 must be c_4. One additional comparison is performed by P_3 to produce the two middle elements of C. When n = 4, we can use two copies of the network in Fig. 3.2 followed by three comparators, as shown in Fig. 3.3 for $A = \{3, 5, 7, 9\}$ and $B = \{2, 4, 6, 8\}$.

In general, an (n, n)-merging network is obtained by the following recursive construction. First, the odd-numbered elements of A and B, that is, $\{a_1, a_3, a_5, \ldots, a_{n-1}\}$ and $\{b_1, b_3, b_5, \ldots, b_{n-1}\}$, are merged using an $(n/2, n/2)$-merging network to produce a sequence $\{d_1, d_2, d_3, \ldots, d_n\}$. Simultaneously, the even-numbered elements of the two sequences, $\{a_2, a_4, a_6, \ldots, a_n\}$ and $\{b_2, b_4, b_6, \ldots, b_n\}$, are also merged using an $(n/2, n/2)$-merging network to produce a sequence $\{e_1, e_2, e_3, \ldots, e_n\}$. The final sequence $\{c_1, c_2, \ldots, c_{2n}\}$ is now obtained from

$$c_1 = d_1, \quad c_{2n} = e_n, \quad c_{2i} = \min(d_{i+1}, e_i), \quad \text{and} \quad c_{2i+1} = \max(d_{i+1}, e_i)$$

$$\text{for } i = 1, 2, \ldots, n - 1.$$

The final comparisons are accomplished by a rank of n − 1 comparators as illustrated in Fig. 3.4. Note that each of the $(n/2, n/2)$-merging networks is constructed by applying the same rule recursively, that is, by using two $(n/4, n/4)$-merging networks followed by a rank of $(n/2) - 1$ comparators.

The merging network in Fig. 3.4 is based on a method known as odd–even merging. That this method works in general is shown as follows. First note that $d_1 = \min(a_1, b_1)$ and $e_n = \max(a_n, b_n)$, which means that c_1 and c_{2n} are computed properly. Now observe that in the sequence $\{d_1, d_2, \ldots, d_n\}$, i elements are smaller than or equal to d_{i+1}. Each of these is an odd-numbered element of either A or B. Therefore, 2i elements of A and B are smaller than or equal to d_{i+1}. In other words,

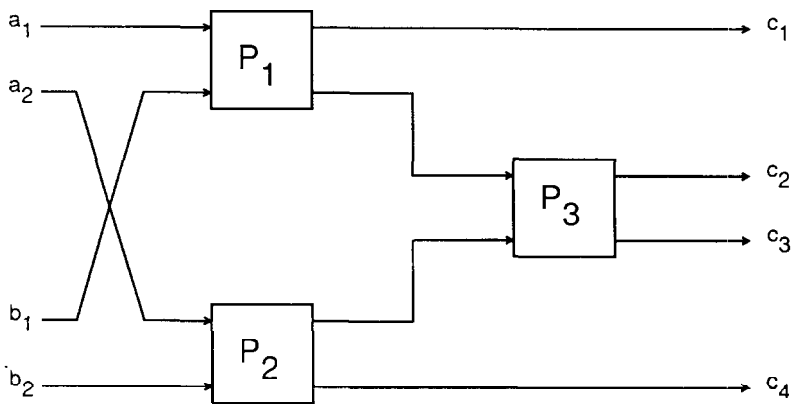


Figure 3.2 Merging two sequences of two elements each.

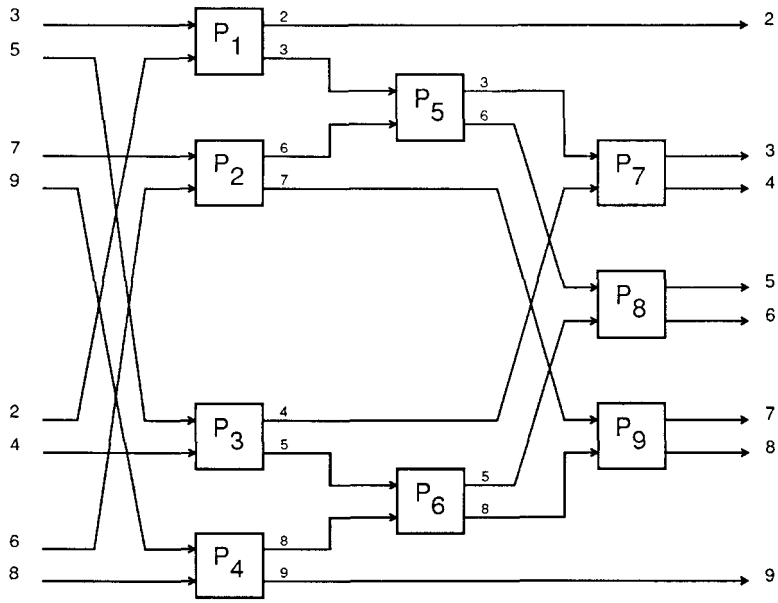


Figure 3.3 Merging two sequences of four elements each.

$d_{i+1} \geqslant c_{2i}$. Similarly, $e_i \geqslant c_{2i}$. On the other hand, in the sequence $\{c_1, c_2, \ldots, c_{2n}\}$, 2i elements from A and B are smaller than or equal to c_{2i+1}. This means that c_{2i+1} is larger than or equal to $(i + 1)$ odd-numbered elements belonging to either A or B. In other words, $c_{2i+1} \geqslant d_{i+1}$. Similarly, $c_{2i+1} \geqslant e_i$. Since $c_{2i} \leqslant c_{2i+1}$, the preceding inequalities imply that $c_{2i} = \min(d_{i+1}, e_i)$, and $c_{2i+1} = \max(d_{i+1}, e_i)$, thus establishing the correctness of odd–even merging.

Analysis. Our analysis of odd–even merging will concentrate on the time, number of processors, and total number of operations required to merge.

(i) Running Time. We begin by assuming that a comparator can read its input, perform a comparison, and produce its output all in one time unit. Now; let $t(2n)$ denote the time required by an (n,n)-merging network to merge two sequences of length n each. The recursive nature of such a network yields the following recurrence for $t(2n)$:

$$t(2) = 1 \qquad \text{for } n = 1 \quad \text{(see Fig. 3.1)},$$

$$t(2n) = t(n) + 1 \qquad \text{for } n > 1 \quad \text{(see Fig. 3.4)},$$

whose solution is easily seen to be $t(2n) = 1 + \log n$. This is significantly faster than the best, namely, $O(n)$, running time achievable on a sequential computer.

(ii) Number of Processors. Here we are interested in counting the number of comparators required to odd–even merge. Let $p(2n)$ denote the number of cornpara-

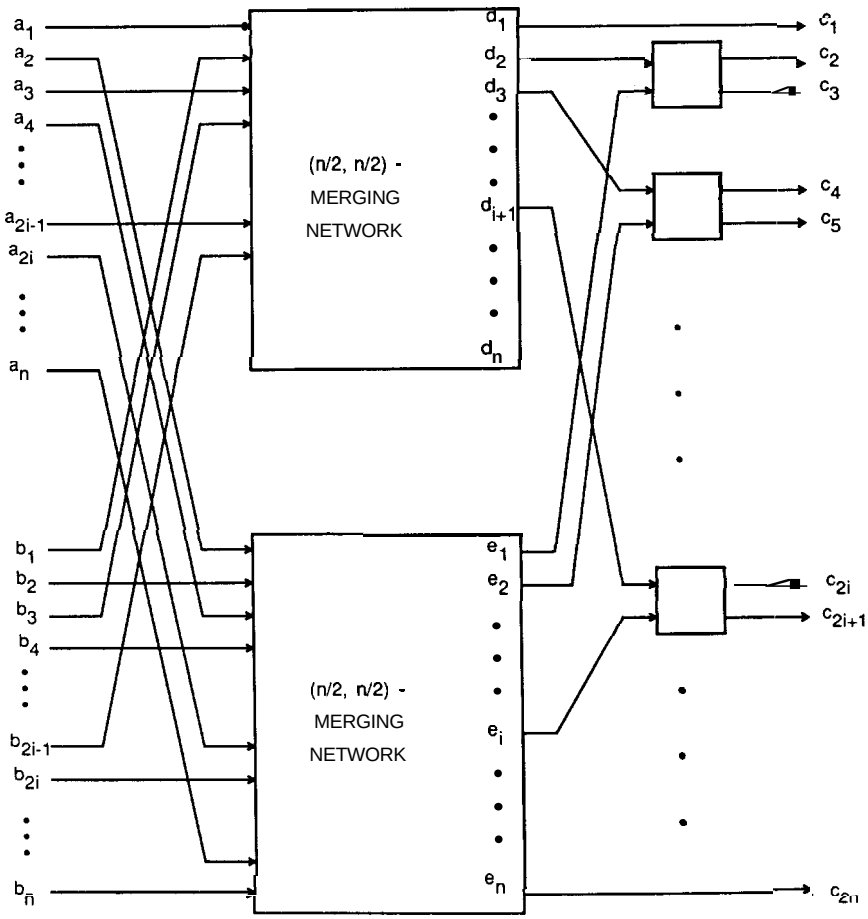


Figure 3.4 Odd–even merging.

tors in an (n, n)-merging network. Again, we have a recurrence:

$$p(2) = 1 \qquad \text{for } n = 1 \quad \text{(see Fig. 3.1)},$$

$$p(2n) = 2p(n) + (n - 1) \qquad \text{for } n > 1 \quad \text{(see Fig. 3.4)},$$

whose solution $p(2n) = 1 + n \log n$ is also straightforward.

(iii) Cost. Since $t(2n) = 1 + \log n$ and $p(2n) = 1 + n \log n$, the total number of comparisons performed by an (n, n)-merging network, that is, the network's cost, is

$$c(2n) = p(2n) \times t(2n)$$

$$= O(n \log^2 n).$$

Our network is therefore not cost optimal as it performs more operations than the $O(n)$ sufficient to merge sequentially.

Discussion. In this section we presented an example of a special-purpose architecture for merging. These merging networks, as we called them, have the following interesting property: The sequence of comparisons they perform is fixed in advance. Regardless of the input, the network will always perform the same number of comparisons in a predetermined order. This is why such networks are sometimes said to be oblivious of their input.

Our analysis showed that the (n, n)-merging network studied is extremely fast, especially when compared with the best possible sequential merging algorithm. For example, it can merge two sequences of length 2^{20} elements each in twenty-one steps; the same result would require more than two million steps on a sequential computer. Unfortunately, such speed is achieved by using an unreasonable number of processors. Again, for $n = 2^{20}$, our (n, n)-merging network would consist of over twenty million comparators! In addition, the architecture of the network is highly irregular, and the wires linking the comparators have lengths that vary with n. This suggests that, although theoretically appealing, merging networks would be impractical for large values of n.

3.3 MERGING ON THE CREW MODEL

Our study of odd–even merging identified a problem associated with merging networks in general, namely, their inflexibility. A fixed number of comparators are assembled in a fixed configuration to merge sequences of fixed size. Although this may prove adequate for some applications, it is desirable in general to have a parallel algorithm that adapts to the number of available processors on the parallel computer at hand. This section describes one such algorithm. In addition to being adaptive, the algorithm is also cost optimal: Its running time multiplied by the number of processors used equals, to within a constant multiplicative factor, the lower bound on the number of operations required to merge. The algorithm runs on the CREW SM SIMD model. It assumes the existence, and makes use of, a sequential procedure for merging two sorted sequences. We therefore begin by presenting this procedure.

3.3.1 Sequential Merging

Two sequences of numbers $A = \{a_1, a_2, \ldots, a_n\}$ and $B = \{b_1, b_2, \ldots, b_s\}$ sorted in nondecreasing order are given. It is required to merge A and B to form a third sequence C, also sorted in nondecreasing order. The merging process is to be performed by a single processor. This can be done by the following algorithm. Two pointers are used, one for each sequence. Initially, the pointers are positioned at elements a_1 and b_1, respectively. The smaller of a_1 and b_1 is assigned to c_1, and the pointer to the sequence from which c_1 came is advanced one position. Again, the two elements pointed to are compared: The smaller becomes c_2 and the pointer to it is advanced. This continues until one of the two input sequences is exhausted; the elements left over in the other sequence are now copied in C. The algorithm is given in

what follows as procedure SEQUENTIAL MERGE. Its description is greatly simplified by assuming the existence of two fictional elements a_{r+}, and b_{s+},, both of which are equal to infinity.

```
procedure SEQUENTIAL MERGE (A, B, C)
Step 1: (1.1) i ← 1
        (1.2) j ← 1.
Step 2: for k = 1 to r + s do
            if a_i < b_j then (i) c_k ← a_i
                              (ii) i t i + 1
                         else (i) c_k ← b_j
                              (ii) j ← j + 1
            end if
        end for. □
```

The procedure takes sequences *A* and B as input and returns sequence *C* as output. Since each comparison leads to one element of C being defined, there are exactly r + *s* such comparisons, and in the worst case, when r = *s* = n, say, the algorithm runs in $O(n)$ time. In view of the $\Omega(n)$ lower bound on merging derived in section 3.1, procedure SEQUENTIAL MERGE is optimal.

3.3.2 Parallel Merging

A CREW SM SIMD computer consists of N processors $P_1, P_2, \ldots, P_N$. It is required to design a parallel algorithm for this computer that takes the two sequences *A* and B as input and produces the sequence C as output, as defined earlier. Without loss of generality, we assume that r ⩽ s.

It is desired that the parallel algorithm satisfy the properties stated in section 2.4, namely, that

(i) the number of processors used by the algorithm be sublinear and adaptive,

(ii) the running time of the algorithm be adaptive and significantly smaller than the best sequential algorithm, and

(iii) the cost be optimal.

We now describe an algorithm that satisfies these properties. It uses N processors where N ⩽ r and in the worst case when r = *s* = n runs in $O((n/N) + \log n)$ time. The algorithm is therefore cost optimal for N ⩽ $n/\log n$. In addition to the basic arithmetic and logic functions usually available, each of the N processors is assumed capable of performing the following two sequential procedures:

1. Procedure SEQUENTIAL MERGE described in section 3.3.1.

2. Procedure BINARY SEARCH described in what follows. The procedure

takes as input a sequence $S = \{s_1, s_2, \ldots, s_n\}$ of numbers sorted in nondecreasing order and a number **x.** If x belongs to S, the procedure returns the index k of an element s_k in S such that $\mathbf{x} = s_k$. Otherwise, the procedure returns a zero. Binary search is based on the divide-and-conquer principle. At each stage, a comparison is performed between x and an element of S. Either the two are equal and the procedure terminates or half of the elements of the sequence under consideration are discarded. The process continues until the number of elements left is 0 or 1, and after at most one additional comparison the procedure terminates.

procedure BINARY SEARCH **(S,** *x, k)*

Step 1: *(1.1)* $i \leftarrow 1$
(1.2) $h \leftarrow n$
(1.3) $k \leftarrow 0$.

Step *2:* **while** $i \leqslant$ h **do**
(2.1) $m \leftarrow \lfloor (i + h)/2 \rfloor$
(2.2) **if** $x = s_m$ **then** (i) $k \leftarrow m$
(ii) $i \leftarrow h + 1$
else if $x < s_m$ **then** $h \leftarrow m - 1$
else $i \leftarrow m + 1$
end if
end if
end while. □

Since the number of elements under consideration is reduced by one-half at each step, the procedure requires $O(\log n)$ time in the worst case.

We are now ready to describe our first parallel merging algorithm for a shared-memory computer. The algorithm is presented as procedure CREW MERGE.

procedure CREW MERGE *(A,* B, C)

Step *1:* {Select N – *1* elements of *A* that subdivide that sequence into N subsequences of approximately the same size. Call the subsequence formed by these N – *1* elements *A'.* A subsequence B' of N – *1* elements of *B* is chosen similarly. This step is executed **as** follows:}
for i = 1 **to** $N - 1$ **do in parallel**
Processor P_i determines a'_i and b'_i from
(1.1) $a'_i \leftarrow a_{i\lceil r/N \rceil}$
(1.2) $b'_i \leftarrow b_{i\lceil s/N \rceil}$

end for.

Step 2: {Merge *A'* and B' into a sequence of triples $V = \{v_1, v_2, \ldots, v_{2N-2}\}$, where each triple consists of an element of *A'* or B' followed by its position in A' or B' followed by the name of its sequence of origin, that is, A or B. This is done as follows:}
(2.1) **for** i = 1 **to** $N - 1$ **do in parallel**
(i) Processor P_i uses BINARY SEARCH on B' to find the smallest j such that $a'_i < b'_j$

(ii) **if** j exists **then** $v_{i+j-1} \leftarrow (a'_i, \text{i}, \text{A})$
else $v_{i+N-1} \leftarrow (a'_i, i, \text{A})$
end if
end for

(2.2) **for** i = 1 **to** N − 1 **do in parallel**
(i) Processor P_i uses BINARY SEARCH on A' to find the **smallest** j such that $b'_i < a'_j$
(ii) **if** j exists **then** $v_{i+j-1} \leftarrow (b'_i, \text{i}, B)$
else $v_{i+N-1} \leftarrow (b'_i, \text{i}, \text{B})$
end if
end for.

Step 3: {Each processor merges and inserts into C the elements of two subsequences, one from A and one from B. The indices of the two elements (one in A and one in B) at which each processor is to begin merging are first computed and stored in an array Q of ordered pairs. This step is executed as follows:)

(3.1) $Q(1) \leftarrow (1, 1)$

(3.2) **for** i = 2 **to** N **do in parallel**
if $v_{2i-2} = (a'_k, \text{k}, \text{A})$ **then** processor P_i
(i) uses BINARY SEARCH on B to find the smallest j such that $b_j > a'_k$
(ii) $Q(i) \leftarrow (k\lceil r/N \rceil, j)$
else processor P_i
(i) uses BINARY SEARCH on A to find the smallest j such that $a_j > b'_k$
(ii) $Q(i) \leftarrow (j, k\lceil s/N \rceil)$
end if
end for

(3.3) for i = 1 **to** N **do in parallel**
Processor P_i uses SEQUENTIAL MERGE and $Q(i) = (x, \text{y})$ to merge two subsequences one beginning at a, and the other at b_y and places the result of the merge in array C beginning at position x + y − 1. The merge continues until
(i) an element larger than or equal to the first component of v_{2i} is encountered in each of A and B (when i ≤ N − 1)
(ii) no elements are left in either A or B (when i = N)
end for. □

Before analyzing the running time of the algorithm, we make the following two observations:

(i) In general instances, an element a_i of A is compared to an element b_j of B to determine which is smaller; if it turns out that $a_i = b_j$, then the algorithm decides arbitrarily that a_i is smaller.

(ii) Concurrent-read operations are performed whenever procedure BINARY SEARCH is invoked, namely, in steps 2.1, 2.2, and 3.2. Indeed, in each of these instances several processors are executing a binary search over the same sequence.

Analysis. A step-by-step analysis of CREW MERGE follows:

Step 1: With all processors operating in parallel, each processor computes two subscripts. Therefore this step requires constant time.

Step *2:* This step consists of two applications of procedure BINARY SEARCH to a sequence of length $N - 1$, each followed by an assignment statement. This takes $O(\log N)$ time.

Step *3:* Step 3.1 consists of a constant-time assignment, and step 3.2 requires at most $O(\log s)$ time. To analyze step **3.3,** we first observe that V contains $2N - 2$ elements that divide C into $2N - 1$ subsequences with maximum size equal to $(\lceil r/N \rceil + \lceil s/N \rceil)$. This maximum size occurs if, for example, one element a'_i of A' equals an element b'_j of B'; then the $\lceil r/N \rceil$ elements smaller than or equal to a'_i (and larger than or equal to a'_{i-1}) are also smaller than or equal to b'_j, and similarly, the $\lceil s/N \rceil$ elements smaller than or equal to b'_j (and larger than or equal to b'_{j-1}) are also smaller than or equal to a'_i. In step 3 each processor creates two such subsequences of C whose total size is therefore no larger than $2(\lceil r/N \rceil + \lceil s/N \rceil)$, except P_n which creates only one subsequence of C. It follows that procedure SEQUENTIAL MERGE takes at most $O((r + s)/N)$ time.

In the worst case, $r = s = n$, and since $n \geqslant$ N, the algorithm's running time is dominated by the time required by step 3. Thus

$$t(2n) = O((n/N) + \log n).$$

Since $p(2n) = N$, $c(2n) = p(2n) \times t(2n) = O(n + N \log n)$, and the algorithm is cost optimal when $N \leqslant n/\log n$.

Example 3.1

Assume that a CREW SM SIMD computer with N = 4 processors is available and it is required to merge $A = \{2, 3, 4, 6, 11, 12, 13, 15, 16, 20, 22, 24\}$ and B = {1, 5, 7, 8, 9, 10, 14, 17, 18, 19, 21, 23}, that is, $r = s = 12$.

The two subsequences $A' = \{4, 12, 16)$ and $B' = \{7, 10, 18)$ are found in step 1 and then merged in step 2 to obtain

$$V = \{(4, 1, A), (7, 1, B), (10, 2, B), (12, 2, A), (16, 3, A), (18, 3, B)\}.$$

In steps 3.1 and 3.2, $Q(1) = (1, 1)$, Q(2) = (5, 3), $Q(3) = (6, 7)$, and $Q(4) = (10, 9)$ are determined. In step 3.3 processor P_1 begins at elements a, = 2 and $b_1 = 1$ and merges all elements of A and B smaller than 7, thus creating the subsequence {1, 2, 3, 4, 5, 6} of C. Similarly, processor P_2 begins at a, = 11 and $b_3 = 7$ and merges all elements smaller than 12, thus creating {7, 8, 9, 10, 11). Processor P_3 begins at a, = 12 and b, = 14 and creates {12, 13, 14, 15, 16, 17). Finally P_4 begins at a,, = 20 and $b_9 = 18$ and creates {18, 19, 20, 21, 22, 23, 24}. The resulting sequence C is therefore {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24). The elements of A' and B' are shown underlined in C. □

3.4 MERGING ON THE EREW MODEL

As we saw in the previous section, concurrent-read operations are performed at several places of procedure CREW MERGE. We now show how this procedure can be adapted to run on an N-processor EREW SM SIMD computer that, by definition, disallows any attempt by more than one processor to read from a memory location. The idea of the adaptation is quite simple: All we have to do is find a way to simulate multiple-read operations. Once such a simulation is found, it can be used by the parallel merge algorithm (and in general by any algorithm with multiple-read operations) to perform every read operation from the EREW memory. Of course, we require the simulation to be efficient. Simply queuing all the requests to read from a given memory location and serving them one after the other is surely inadequate: It can increase the running time by a factor of N in the worst case. On the other hand, using procedure BROADCAST of chapter 2 is inappropriate: A multiple-read operation from a memory location may not necessarily involve all processors. Typically, several arbitrary subsets of the set of processors attempt to gain access to different locations, one location per subset. In chapter 1 we described a method for performing the simulation in this general case. This is now presented more formally as procedure MULTIPLE BROADCAST in what follows.

Assume that an algorithm designed to run on a CREW SM SIMD computer requires a total of M locations of shared memory. In order to simulate this algorithm on the EREW model with N processors, where $N = 2^q$ for $q \geq 1$, we increase the size of the memory from M to $M(2N - 1)$. Thus, each of the M locations is thought of as the root of a binary tree with N leaves. Such a tree has $q + 1$ levels and a total of $2N - 1$ nodes, as shown in Fig. 3.5 for $N = 16$. The nodes of the tree represent consecutive locations in memory. Thus if location D is the root, then its left and right children are $D + 1$ and $D + 2$, respectively. In general, the left and right children of $D + x$ are $D + 2x + 1$ and $D + 2x + 2$, respectively.

Assume that processor P_i wishes at some point to read from some location $d(i)$ in memory. It places its request at location $d(i) + (N - 1) + (i - 1)$, a leaf of the tree rooted at $d(i)$. This is done by initializing two variables local to P_i:

1. level(i), which stores the current level of the tree reached by P_i's request, is initialized to 0, and
2. loc(i), which stores the current node of the tree reached by P_i's request, is initialized to $(N - 1) + (i - 1)$. Note that P_i need only store the position in the tree relative to $d(i)$ that its request has reached and not the actual memory location $d(i) + (N - 1) + (i - 1)$.

The simulation consists of two stages: the ascent stage and the descent stage. During the ascent stage, the processors proceed as follows: At each level a processor P_i occupying a left child is first given priority to advance its request one level up the tree.

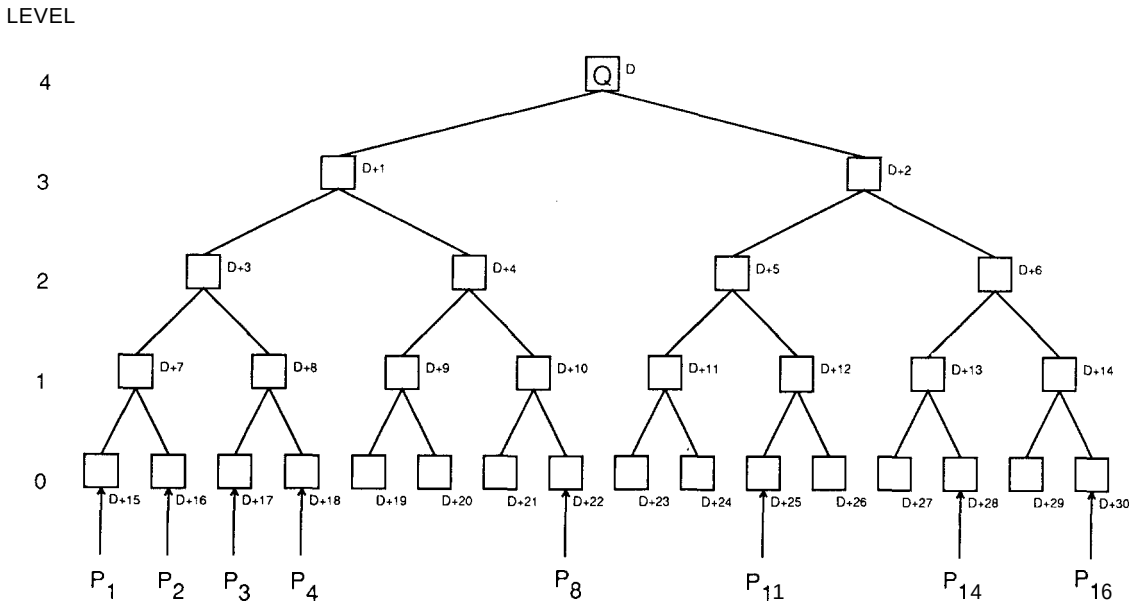


Figure 3.5 Memory organization for **multiple** broadcasting.

It does so by marking the parent location with a special marker, say, **[i].** It then updates its level and location. In this case, a request at the right child is immobilized for the remainder of the procedure. Otherwise (i.e., if there was no processor occupying the left child) a processor occupying the right child can now "claim" the parent location. This continues until at most two processors reach level (log N) − 1. They each in turn read the value stored in the root, and the descent stage commences. The value just read goes down the tree of memory locations until every request to read by a processor has been honored. Procedure MULTIPLE BROADCAST follows.

procedure MULTIPLE BROADCAST $(d(1), d(2), \ldots, d(N))$

Step 1: **for** i = 1 **to** N **do in parallel**
{P_i initializes level(i) and loc(i)}
(1.1) level(i)←0
(1.2) loc(i) ← N + i − 2
(1.3) store [i] in location $d(i)$ + loc(i)
end for.

Step 2: **for** v = 0 **to** (log N) − 2 **do**
(2.1) **for** i = 1 **to** N **do in parallel**
{P_i at a left child advances up its tree}
(2.1.1) $x \leftarrow \lfloor (\text{loc}(i) - 1)/2 \rfloor$
(2.1.2) if loc(i) is odd and level(i)=v

```
                then (i) loc(i) ← x
                     (ii) store [i] in location d(i) + loc(i)
                     (iii) level(i) ← level(i) + 1
                end if
            end for
        (2.2) for i = 1 to N do in parallel
            {P_i at a right child advances up its tree if possible}
            if d(i) + x does not already contain a marker [j] for some 1 ≤ j ≤ N
            then (i) loc(i) ← x
                 (ii) store [i] in location d(i) + loc(i)
                 (iii) level(i) ← level(i) + 1
            end if
        end for
    end for.

Step 3: for v = (log N) − 1 down to 0 do
        (3.1) for i = 1 to N do in parallel
            {P_i at a left child reads from its parent and then moves down the tree}
            (3.1.1) x ← ⌊(loc(i) − 1)/2⌋
            (3.1.2) y ← (2 x loc(i)) + 1
            (3.1.3) if loc(i) is odd and level(i) = v
                then (i) read the contents of d(i) + x
                     (ii) write the contents of d(i) + x in location
                          d(i) + loc(i)
                     (iii) level(i) ← level(i) − 1
                     (iv) if location d(i) + y contains [i]
                          then loc(i) ← y
                          else loc(i) ← y + 1
                          end if
                end if
            end for
        (3.2) for i = 1 to N do in parallel
            {P_i at a right child reads from its parent and then moves down the tree}
            if loc(i) is even and level(i)=v
            then (i) read the contents of d(i) + x
                 (ii) write the contents of d(i) + x in location d(i) + loc(i)
                 (iii) level(i) ← level(i) − 1
                 (iv) if location d(i) + y contains [i]
                      then loc(i) ← y
                      else loc(i) ← y + 1
                      end if
            end if
        end for
    end for. □
```

Step 1 of the procedure consists of three constant-time operations. Each of the ascent and descent stages in steps 2 and 3, respectively, requires $O(\log N)$ time. The overall running time of procedure MULTIPLE BROADCAST is therefore $O(\log N)$.

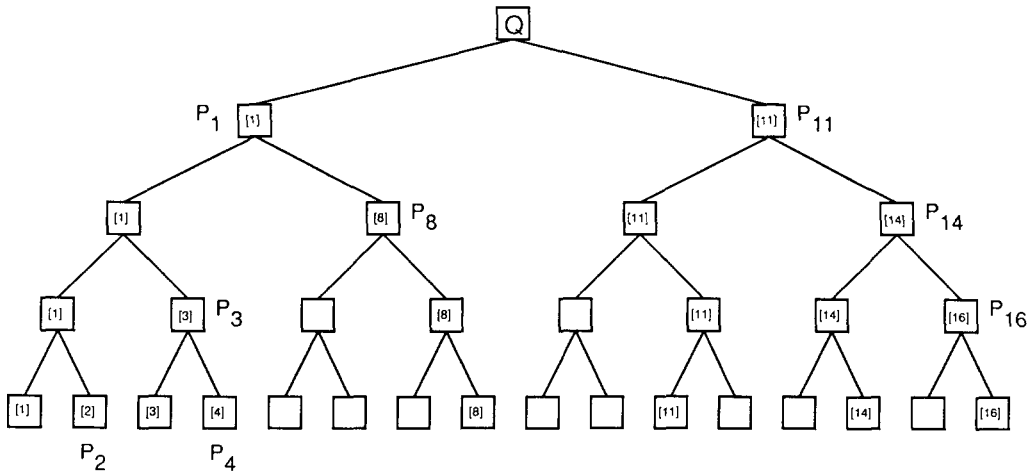


Figure 3.6 Memory contents after step 2 of procedure MULTIPLE BROADCAST.

Example 3.2

Let N = 16 and assume that at a given moment during the execution of a CREW parallel algorithm processors P_1, P_2, P_3, P_4, P_8, P_{11}, P_{14}, and P_{16} need to read a quantity Q from a location D in memory. When simulating this multiple-read operation on an EREW computer using MULTIPLE BROADCAST, the processors place their requests at the appropriate leaves of a tree of locations rooted at D during step 1, as shown in Fig. 3.5. Figure *3.6* shows the positions of the various processors and the contents of memory locations at the end of step 2. The contents of the memory locations at the end of step 3 are shown in Fig. **3.7.** □

Note that:

1. The markers [i] are chosen so that they can be easily distinguished from data values such as Q.
2. If during a multiple-read step of the CREW algorithm being simulated, a processor P_i does not wish to read from memory, then $d(i)$ may be chosen arbitrarily among the M memory locations used by the algorithm.
3. When the procedure terminates, the value of level(i) is negative and that of loc(i) is out of bounds. These values are meaningless. This is of no consequence, however, since level(i) and loc(i) are always initialized in step 1.

We are now ready to analyze the running time $t(2n)$ of an adaptation of procedure CREW MERGE for the EREW model. Since every read operation (simple or multiple) is simulated using procedure MULTIPLE BROADCAST in $O(\log N)$ time, the adapted procedure is at most $O(\log N)$ times slower than procedure CREW

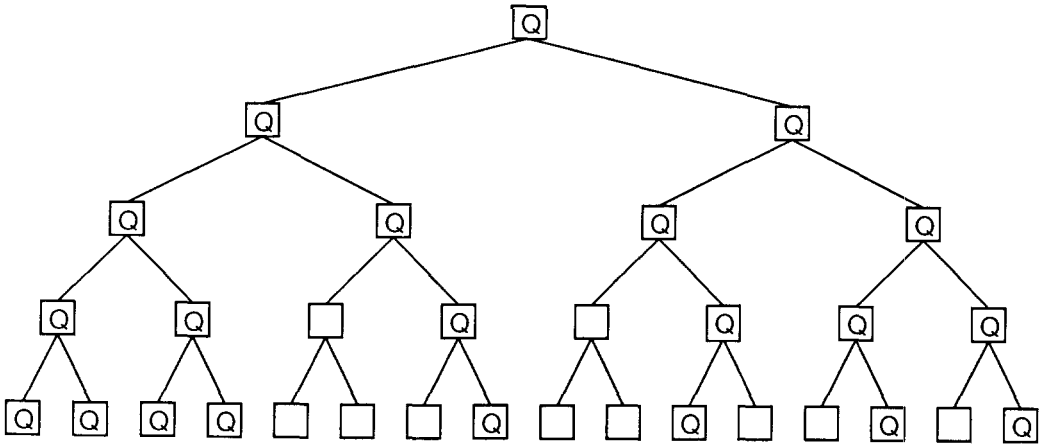


Figure 3.7 Memory contents at end of procedure MULTIPLE BROADCAST.

MERGE, that is,

$$t(2n) = O(\log \mathrm{N}) \times O(n/N + \log \mathrm{n})$$
$$= O((n/N)\log n + \log^2 n).$$

The algorithm has a cost of

$$c(2n) = \mathrm{O}(\mathrm{n} \log \mathrm{n} + \mathrm{N} \log^2 \mathrm{n})$$

which is not optimal. Furthermore, since procedure CREW MERGE uses $O(n)$ locations of shared memory, the storage requirements of its adaptation for the EREW model are $O(Nn)$. In the following section an algorithm for merging on the EREW model is described that is cost optimal and uses only $O(n)$ shared-memory locations.

3.5 A BETTER ALGORITHM FOR THE EREW MODEL

We saw in the previous section how a direct simulation of the CREW merging algorithm on the EREW model is not cost optimal. This is due to the logarithmic factor always introduced by procedure MULTIPLE BROADCAST. Clearly, in order to match the performance of procedure CREW MERGE, another approach is needed. In this section we describe an adaptive and cost-optimal parallel algorithm for merging on the EREW SM SIMD model of computation. The algorithm merges two sorted sequences $A = (\mathrm{a},, a_2, \ldots, \mathrm{a},)$ and $\mathrm{B} = \{b_1, b_2, \ldots, b_s\}$ into a single sequence $\mathrm{C} = \{c_1, c_2, \ldots, c_{r+s}\}$. It uses N processors $P_1, P_2, \ldots, P_N$, where $1 \leqslant \mathrm{N} \leqslant \mathrm{r} + s$ and, in the worst case when $\mathrm{r} = s = \mathrm{n}$, runs in $O((n/N) + \log \mathrm{N} \log \mathrm{n})$ time. A building block of the algorithm is a sequential procedure for finding the median of two sorted sequences. This procedure is presented in section 3.5.1. The merging algorithm itself is the subject of section 3.5.2.

3.5.1 Finding the Median of Two Sorted Sequences

In this section we study a variant of the selection problem visited in chapter 2. Given two sorted sequences $A = \{a_1, a_2, \ldots, a_r\}$ and $B = \{b_1, b_2, \ldots, b_s\}$, where $r, s \geq 1$, let $A.B$ denote the sequence of length $m = r + s$ resulting from merging A and B. It is required to find the median, that is, the $\lceil m/2 \rceil$th element, of $A.B$. Without actually forming $A.B$, the algorithm we are about to describe returns a pair (a_x, b_y) that satisfies the following properties:

1. Either a_x or b_y is the median of $A.B$, that is, either a_x or b_y is larger than precisely $\lceil m/2 \rceil - 1$ elements and smaller than precisely $\lfloor m/2 \rfloor$ elements.
2. If a_x is the median, then b_y is either
   (i) the largest element in B smaller than or equal to a_x, or
   (ii) the smallest element in B larger than or equal to a_x.
   Alternatively, if b_y is the median, then a_x is either
   (i) the largest element in A smaller than or equal to b_y or
   (ii) the smallest element in A larger than or equal to b_y.
3. If more than one pair satisfies 1 and 2, then the algorithm returns the pair for which $x + y$ is smallest.

We shall refer to (a_x, b_y) as the *median pair* of $A.B$. Thus x and y are the *indices of the median pair*. Note that a_x is the median of $A.B$ if either

(i) $a_x > b_y$ and $x + y - 1 = \lceil m/2 \rceil - 1$ or
(ii) $a_x < b_y$ and $m - (x + y - 1) = \lfloor m/2 \rfloor$.

Otherwise b_y is the median of $A.B$.

Example 3.3

Let $A = \{2, 5, 7, 10\}$ and $B = \{1, 4, 8, 9\}$ and observe that the median of $A.B$ is 5 and belongs to A. There are two median pairs satisfying properties 1 and 2:

(i) $(a_2, b_2) = (5, 4)$, where 4 is the largest element in B smaller than or equal to 5;
(ii) $(a_2, b_3) = (5, 8)$, where 8 is the smallest element in B larger than or equal to 5.
The median pair is therefore (5, 4). □

The algorithm, described in what follows as procedure TWO-SEQUENCE MEDIAN, proceeds in stages. At the end of each stage, some elements are removed from consideration from both A and B. We denote by n_A and n_B the number of elements of A and B, respectively, still under consideration at the beginning of a stage and by w the smaller of $\lfloor n_A/2 \rfloor$ and $\lfloor n_B/2 \rfloor$. Each stage is as follows: The medians a and b of the elements still under consideration in A and in B, respectively, are compared. If $a \geq b$, then the largest (smallest) w elements of $A(B)$ are removed from consideration. Otherwise, that is, if $a < b$, then the smallest (largest) w elements of $A(B)$ are removed

from consideration. This process is repeated until there is only one element left still under consideration in one or both of the two sequences. The median pair is then determined from a small set of candidate pairs. The procedure keeps track of the elements still under consideration by using two pointers to each sequence: low, and high, in A and low, and high, in B.

procedure TWO-SEQUENCE MEDIAN (A, B, **x,** y)

Step 1: (1.1) $\mathbf{low}_A \leftarrow 1$
(1.2) low, $\leftarrow 1$
(1.3) high, $\leftarrow r$
(1.4) high, $\leftarrow s$
(1.5) $n_A \leftarrow r$
(1.6) $n_B \leftarrow$ s.

Step 2: **while** n, > 1 **and** n, > 1 do
(2.1) $u \leftarrow \mathbf{low}_A + \lceil(\mathbf{high}_A - \mathbf{low}_A - 1)/2\rceil$
(2.2) $v \leftarrow \mathbf{low}_B + \lceil(\mathbf{high}_B -$ low, $- 1)/2\rceil$
(2.3) $w \leftarrow \min(\lfloor n_A/2 \rfloor, \lfloor n_B/2 \rfloor)$
(2.4) $n_A \leftarrow$ n, $- w$
(2.5) $n_B \leftarrow n_B - w$
(2.6) **if** a, $\geqslant$ b,
then (i) high, $\leftarrow$ high, $- w$
(ii) low, $\leftarrow$ low, $+ w$
else (i) low, $\leftarrow$ low, $+ w$
(ii) $\mathbf{high}_B \leftarrow \mathbf{high}_B - w$
end if
end while.

Step 3: Return as **x** and y the indices of the pair from $\{a_{u-1}$, a,, $a_{m+1}\}$ x $\{b_{v-1}$, b,, $b_{v+1}\}$ satisfying properties 1–3 of a median pair. □

Note that procedure TWO-SEQUENCE MEDIAN returns the indices of the median pair (a_x, b_y) rather than the pair itself.

Example 3.4

Let A = {10, 11, 12, 13, 14, 15, 16, 17, 18) and B = {3, 4, 5, 6, 7, 8, 19, 20, 21, 22}. The following variables are initialized during step 1 of procedure **TWO-SEQUENCE** MEDIAN: low, = low, = 1, high, $= n_A = 9$, and high, = n, = 10.

In the first iteration of step 2, $u = v = 5$, $w = \min(4, 5) = 4$, $n_A = 5$, and n, = 6. Since a, > b,, high, =low,= 5. In the second iteration, $u = 3$, $v = 7$, $w = \min(2, 3) = 2$, n, = 3, and n, = 4. Since a, $< b_7$, low, = 3 and high, = 8. In the third iteration, u = 4, $v = 6$, $w = \min(1, 2) = 1$, $n_A = 2$, and n, = 3. Since a, $> b_6$, high, = 4 and low, = 6. In the fourth and final iteration of step 2, u = 3, $v = 7$, $w = \min(1, 1) = 1$, $n_A = 1$, and n, = 2. Since a, $< b_7$, $\mathbf{low}_A = 4$ and high, = 7.

In step 3, two of the nine pairs in {11, 12, 13) x {8, 19, 20) satisfy the first two properties of a median pair. These pairs are $(a_4, b_6) = (13, 8)$ and $(a_4, b_7) = (13, 19)$. The procedure thus returns (4, 6) as the indices of the median pair. □

Analysis. Steps 1 and 3 require constant time. Each iteration of step 2 reduces the smaller of the two sequences by half. For constants c_1 and c_2 procedure TWO-SEQUENCE MEDIAN thus requires $c_1 + c_2 \log(\min\{r, s\})$ time, which is $O(\log n)$ in the worst case.

3.5.2 Fast Merging on the EREW Model

We now make use of procedure TWO-SEQUENCE MEDIAN to construct a parallel merging algorithm for the EREW model. The algorithm, presented in what follows as procedure EREW MERGE, has the following properties:

1. It requires a number of processors that is sublinear in the size of the input and adapts to the actual number of processors available on the EREW computer.
2. Its running time is small and varies inversely with the number of processors used.
3. Its cost is optimal.

Given two sorted sequences $A = \{a_1, a_2, \ldots, a_r\}$ and $B = \{b_1, b_2, \ldots, b_s\}$, the algorithm assumes the existence of N processors $P_1, P_2, \ldots, P_N$, where N is a power of 2 and $1 \leqslant N \leqslant r + s$. It merges A and B into a sorted sequence $C = \{c_1, c_2, \ldots, c_{r+s}\}$ in two stages as follows:

Stage 1: Each of the two sequences A and B is partitioned into N (possibly empty) subsequences $A_1, A_2, \ldots, A_N$ and $B_1, B_2, \ldots, B_N$ such that

(i) $|A_i| + |B_i| = (r + s)/N$ for $1 \leqslant i \leqslant N$ and
(ii) all elements in $A_i.B_i$ are smaller than or equal to all elements in $A_{i+1}.B_{i+1}$ for $1 \leqslant i \leqslant N$.

Stage 2: All pairs A_i and B_i, $1 \leqslant i \leqslant N$, are merged simultaneously and placed in C.

The first stage can be implemented efficiently with the help of procedure TWO-SEQUENCE MEDIAN. Stage 2 is carried out using procedure SEQUENTIAL MERGE. In the following procedure $A[i,j]$ is used to denote the subsequence $\{a_i, a_{i+1}, \ldots, a_j\}$ of A if $i \leqslant j$; otherwise $A[i,j]$ is empty. We define $B[i,j]$ similarly.

```
procedure EREW MERGE (A, B, C)

Step 1: (1.1) Processor P_1 obtains the quadruple (1, r, 1, s)
        (1.2) for j = 1 to log N do
                  for i = 1 to 2^(j-1) do in parallel
                  Processor P_i having received the quadruple (e, f, g, h)
                  (1.2.1) {Finds the median pair of two sequences)
                          TWO-SEQUENCE MEDIAN (A[e, f], B[g, h], x, y)
```

(1.2.2) {Computes four pointers p_1, p_2, q_1, and q_2 as follows:}
if a, is the median
then (i) $p_1 \leftarrow x$
(ii) $q_1 \leftarrow$ x + 1
(iii) if $b_y \leqslant$ a, then (a) $p_2 \leftarrow y$
(b) $q_2 \leftarrow y + 1$
else (a) $p_2 \leftarrow$ y − 1
(b) $q_2 \leftarrow y$
end if
else (i) $p_2 \leftarrow y$
(ii) $q_2 \leftarrow$ y + 1
(iii) if a, $\leqslant b_y$ then (a) $p_1 \leftarrow$ x
(b) $q_1 \leftarrow x + 1$
else (a) $p_1 \leftarrow x - 1$
(b) $q_1 \leftarrow x$
end if
end if
(1.2.3) Communicates the quadruple (e, p_1, g, p_2) to P_{2i-1}
(1.2.4) Communicates the quadruple (q_1, f, q_2, h) to P_{2i}
end for
end for.

Step 2: for i = 1 to N do in **parallel**
Processor P_i having received the quadruple (a, b, c, d)
(2.1) w ← 1 + $((i - 1)(r + s))/N$
(2.2) z ← min{$i(r + s)/N$, (r + s)}
(2.3) SEQUENTIAL MERGE ($A[a, b]$, $B[c, d]$, $C[w, z]$)
end for. □

It should be clear that at any time during the execution of the procedure the subsequences on which processors are working are all disjoint. Hence, no concurrent-read operation is ever needed.

Example 3.5

Let A = {10, 11, 12, 13, 14, 15, 16, 17, 18}, B = {3, 4, 5, 6, 7, 8, 19, 20, 21, 22}, and N = 4.

In step 1.1 processor P_1 receives (1, 9, 1, 10). During the first iteration of step 1.2 processor P_1 determines the indices of the median pair of A and B, namely, (4, 6). It keeps (1, 4, 1, 6) and communicates (5, 9, 7, 10) to P_2. During the second iteration, P_1 computes the indices of the median pair of $A[1, 4] = \{10, 11, 12, 13\}$ and $B[1, 6] = \{3, 4, 5, 6, 7, 8\}$, namely, 1 and 5. Simultaneously, P_2 does the same with $A[5, 9] = \{14, 15, 16, 17, 18\}$ and $B[7, 10] = \{19, 20, 21, 22\}$ and obtains 9 and 7. Processor P_1 keeps (1, 0, 1, 5) and communicates (1, 4, 6, 6) to P_2. Similarly, P_2 communicates (5, 9, 7, 6) to P_3 and (10, 9, 7, 10) to P_4.

In step 2, processors P_1 to P_4 simultaneously create $C[1, 19]$ as follows. Having last received (1, 0, 1, 5), P_1 computes $w = 1$ and z = 5 and copies $B[1, 5] = \{3, 4, 5, 6, 7\}$ into $C[1, 5]$. Similarly, P_2, having last received (1, 4, 6, 6), computes w = 6 and z = 10 and merges $A[1, 4]$ and $B[6, 6]$ to obtain $C[6, 10] = \{8, 10, 11, 12, 13\}$. Processor P_3, having last received (5, 9, 7, 6), computes w = 11 and z = 15 and copies

$A[5, 9] = \{14, 15, 16, 17, 18\}$ into $C[11, 15]$. Finally P_4, having last received (10, 9, 7, 10), computes w = 16 and z = 19 and copies $B[7, 10] = \{19, 20, 21, 22\}$ into $C[16, 19]$. □

Analysis. In order to analyze the time requirements of procedure EREW MERGE, note that in step 1.1 processor P_1 reads from memory in constant time. During the jth iteration of step 1.2, each processor involved has to find the indices of the median pair of $(r + s)/2^{j-1}$ elements. This is done using procedure TWO-SEQUENCE MEDIAN in $O(\log[(r + s)/2^{j-1}])$ time, which is $O(\log(r + s))$. The two other operations in step 1.2 take constant time as they involve communications among processors through the shared memory. Since there are log N iterations of step 1.2, step 1 is completed in $O(\log N \times \log(r + s))$ time.

In step 2 each processor merges at most $(r + s)/N$ elements. This is done using procedure SEQUENTIAL MERGE in $O((r + s)/N)$ time. Together, steps 1 and 2 take $O((r + s)/N + \log N \times \log(r + s))$ time. In the worst case, when $r = s = n$, the time required by procedure EREW MERGE can be expressed as

$$t(2n) = O(n/N + \log^2 n),$$

yielding a cost of $c(2n) = O(n + N \log^2 n)$. In view of the $\Omega(n)$ lower bound on the number of operations required to merge, this cost is optimal when $N \leqslant n/\log^2 n$.

3.6 PROBLEMS

3.1 The odd–even merging network described in section 3.2 is just one example from a wide class of merging networks. Show that, in general, any (r, s)-merging network built of comparators must require $\Omega(\log(r + s))$ time in order to completely merge two sorted sequences of length r and s, respectively.

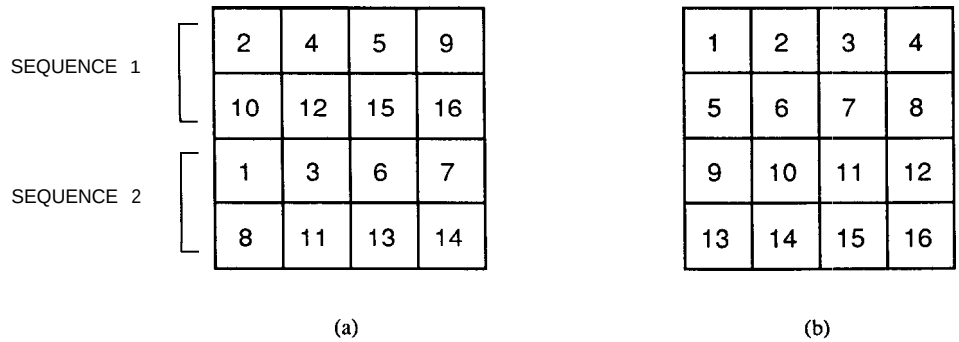
3.2 Show that, in general, any (r, s)-merging network must require $\Omega(s \log r)$ comparators when $r \leqslant s$.

3.3 Use the results in problems 3.1 and 3.2 to draw conclusions about the running time and number of comparators needed by the (n, n) odd–even merging network of section 3.2.

3.4 The odd–even merging network described in section 3.2 requires the two input sequences to be of equal length n. Modify that network so it becomes an (r, s)-merging network, where r is not necessarily equal to s.

3.5 The sequence of comparisons in the odd–even merging network can be viewed as a parallel algorithm. Describe an implementation of that algorithm on an SIMD computer where the processors are connected to form a linear array. The two input sequences to be merged initially occupy processors P, to P, and P_{r+1} to P_s, respectively. When the algorithm terminates, P_i should contain the ith smallest element of the output sequence.

3.6 Repeat problem 3.5 for an m x m mesh-connected SIMD computer. Here the two sequences to be merged are initially horizontally adjacent, that is, one sequence occupies the upper part of the mesh and the second the lower part, as shown in Fig. 3.8(a). The output should be returned, as in Fig. 3.8(b), that is, in row-major order: The ith element resides in row j and column k, where $i = jm + k + 1$. Note that for simplicity, only the processors and their contents are shown in the figure, whereas the communications links have been omitted.



Fipe 3.8 Merging two horizontal sequences on mesh-connected SIMD computer.

3.7 Repeat problem 3.6 for the case where the two input sequences are initially vertically adjacent, that is, one sequence occupies the left part of the mesh and the second the right part, as shown in Fig. 3.9. The result of the merge should appear as in Fig. 3.8(b).

3.8 A sequence {a,, a,, ..., a_{2n}} is said to be bitonic if either
(i) there is an integer $1 \leqslant j \leqslant 2n$ such that

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_j \geqslant a_{j+1} \geqslant \cdots \geqslant a_{2n}$$

or

(ii) the sequence does not initially satisfy condition (i) but can be shifted cyclically until condition (i) is satisfied.
For example, {2, 5, 8, 7, 6, 4, 3, 1} is a bitonic sequence as it satisfies condition (i). Similarly, the sequence {2, 1, 3, 5, 6, 7, 8, 4}, which does not satisfy condition (i), is also bitonic as it can be shifted cyclically to obtain {1, 3, 5, 6, 7, 8, 4, 2}. Let {$a_1, a_2, \ldots$,a,,} be a bitonic sequence and let $d_i = \min\{a_i, a_{n+i}\}$ and $e_i = \max\{a_i$, a,,} for $1 \leqslant i \leqslant n$. Show that
(a) {$d_1, d_2, \ldots, d_n$} and {$e_1, e_2, \ldots, e_n$} are each bitonic and
(b) $\max\{d_1, d_2, \ldots, d_n\} \leqslant \min\{e_1, e_2, \ldots, e_n\}$.

3.9 Two sequences A = {a,, a,,. ..,a,} and B = (a,,,, a,,,, ...,a,,} are given that when concatenated form a bitonic sequence {a,, a,, ..., a,,}. Use the two properties of bitonic sequences derived in problem 3.8 to design an (n, n)-merging network for merging A and *B*.

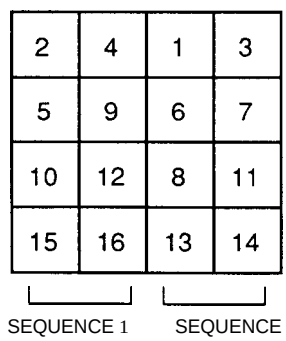


Figure 3.9 Merging two vertical sequences on mesh-connected SIMD computer.

Analyze the running time and number of comparators required. How does your network compare with odd–even merging in those respects?

3.10 Is it necessary for the bitonic merging network in problem 3.9 that the two input sequences be of equal length?

3.11 The sequence of comparisons in the bitonic merging network can be viewed as a parallel algorithm. Repeat problem 3.5 for this algorithm.

3.12 Repeat problem **3.6** for the bitonic merging algorithm.

3.13 Repeat problem **3.7** for the bitonic merging algorithm.

3.14 Design an algorithm for merging on a tree-connected SIMD computer. The two input sequences to be merged, of length r and s, respectively, are initially distributed among the leaves of the tree. Consider the two following situations:

(i) The tree has at least $r + s$ leaves; initially leaves $1, \ldots, r$ store the first sequence and leaves $r + 1, \ldots, r + s$ store the second sequence, one element per leaf.

(ii) The tree has fewer than $r + s$ leaves; initially, each leaf stores a subsequence of the input.

Analyze the running time and cost of your algorithm.

3.15 The running time analysis in problem **3.14** probably indicates that merging on the tree is no faster than procedure SEQUENTIAL MERGE. Show how merging on the tree can be more appealing than sequential merging when several pairs of sequences are queued for merging.

3.16 Consider the following variant of a tree-connected SIMD computer. In addition to the edges of the tree, two-way links connect processors at the same level (into a linear array), as shown in Fig. 3.10 for a four-leaf tree computer. Assume that such a parallel computer, known as a pyramid, has n processors at the base storing two sorted sequences of total length n, one element per processor. Show that $\Omega(n/\log n)$ is a lower bound on the time required for merging on the pyramid.

3.17 Develop a parallel algorithm for merging two sequences of total length n on a pyramid with n base processors. Analyze the running time of your algorithm.

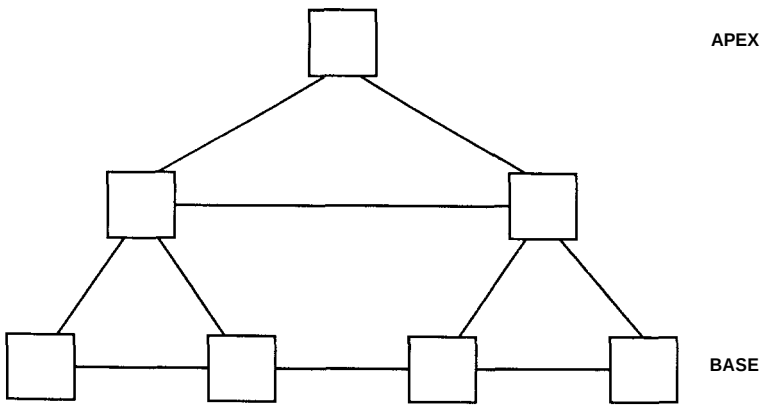


Figure 3.10 Processor pyramid.

3.18 Procedure CREW MERGE assumes that N, the number of processors available to merge two sequences of length r and s, respectively, is smaller than or equal to r when $r \leqslant s$. Modify the procedure so it can handle the case when $r < N \leqslant s$.

3.19 Modify procedure CREW MERGE to use $N \geqslant s \geqslant r$ processors. Analyze the running time and cost of the modified procedure.

3.20 Show that procedure CREW MERGE can be simulated on an EREW computer in $O((n/N) + \log^2 n)$ time if a way can be found to distinguish between simple read operations (each processor needs to gain access to a different memory location) and multiple-read operations.

3.21 Establish the correctness of procedure TWO-SEQUENCE MEDIAN.

3.22 Modify procedure TWO-SEQUENCE MEDIAN so that given two sequences A and B of length r and s, respectively, and an integer $1 \leqslant k \leqslant r + s$, it returns the kth smallest element of A.B. Show that the running time of the new procedure is the same as that of procedure TWO-SEQUENCE MEDIAN.

3.23 Establish the correctness of procedure EREW MERGE.

3.24 Procedure EREW MERGE assumes that N, the number of processors available, is a power of 2. Can you modify the procedure for the case where N is not a power of 2?

3.25 Can the range of cost optimality of procedure EREW MERGE, namely, $N \leqslant n/\log^2 n$, be expanded to, say, $N \leqslant n/\log n$?

3.26 Can procedure EREW MERGE be modified (or a totally new algorithm for the EREW model be developed) to match the $O((n/N) + \log n)$ running time of procedure CREW MERGE?

3.27 Using the results in problems 1.6 and 1.10, show that an algorithm for an N-processor EREW SM SIMD computer requiring $O(N)$ locations of shared memory and time T can be simulated on a cube-connected network with the same number of processors in time $T \times O(\log^2 N)$.

3.28 Analyze the memory requirements of procedure EREW MERGE. Then, assuming that $N = r + s$, use the result in problem 3.27 to determine whether the procedure can be simulated on a cube with N processors in $O(\log^4 N)$ time.

3.29 Assume that $r + s$ processors are available for merging two sequences A and B of length r and s, respectively, into a sequence C. Now consider the following simpler variant of procedure CREW MERGE.

```
for i = 1 to r + s do in parallel
    P_i finds the ith smallest element of A.B (using the procedure in problem 3.22) and places it
    in the ith position of C
end for.
```

Analyze the running time and cost of this procedure.

3.30 Adapt the procedure in problem 3.29 for the case where N processors are available, where $N < r + s$. Compare the running time and cost of the resulting procedure to those of procedure CREW MERGE.

3.31 Develop a parallel merging algorithm for the CRCW model.

3.32 Show how each of the parallel merging algorithms studied in this chapter can lead to a parallel sorting algorithm.

2

Selection

2.1 INTRODUCTION

Our study of parallel algorithm design and analysis begins by addressing the following problem: Given a sequence S of n elements and an integer k, where $1 \leqslant k \leqslant n$, it is required to determine the kth smallest element in S. This is known as the selection problem. It arises in many applications in computer science and statistics. Our purpose in this chapter is to present a parallel algorithm for solving this problem on the shared-memory SIMD model. The algorithm will be designed to meet a number of goals, and our analysis will then confirm that these goals have indeed been met.

We start in section 2.2 by defining the selection problem formally and deriving a lower bound on the number of steps required for solving it on a sequential computer. This translates into a lower bound on the cost of any parallel algorithm for selection. In section 2.3 an optimal sequential algorithm is presented. Our design goals are stated in section 2.4 in the form of properties generally desirable in any parallel algorithm. Two procedures that will be often used in this book are described in section 2.5. Section 2.6 contains the parallel selection algorithm and its analysis.

2.2 THE PROBLEM AND A LOWER BOUND

The problems studied in this and the next two chapters are intimately related and belong to a family of problems known as comparison problems. These problems are usually solved by comparing pairs of elements of an input sequence. In order to set the stage for our presentation we need the following definitions.

2.2.1 Linear Order

The elements of a set A are said to satisfy a linear order $<$ if and only if

(i) for any two elements a and b of A, $a < b$, $a = b$, or $b < a$, and
(ii) for any three elements a, b, and c of A, if $a < b$ and $b < c$, then $a < c$.

The symbol $<$ is to be read "precedes." An example of a set satisfying a linear order is the set of all integers. Another example is the set of letters of the Latin alphabet. We shall say that these sets are linearly ordered. Note that when the elements of A are numbers, it is customary to use the symbol $\leqslant$ to denote "less than or equal to."

2.2.2 Rank

For a sequence $S = \{s_1, s_2, \ldots, s_n\}$ whose elements are drawn from a linearly ordered set, the rank of an element s_i of S is defined as the number of elements in S preceding s_i plus 1. Thus, in $S = \{8, -3, 2, -5, 6, 0\}$ the rank of 0 is 3. Note that if $s_i = s_j$ then s_i precedes s_j if and only if $i < j$.

2.2.3 Selection

A sequence $S = \{s,, s_2, \ldots, s_n\}$ whose elements are drawn from a linearly ordered set and an integer k, where $1 \leqslant k \leqslant n$, are given. It is required to determine the element with rank equal to k. Again, in $S = \{8, -3, 2, -5, 6, 0\}$ the element with rank 4 is 2. We shall denote the element with rank k by $s_{(k)}$.

In the ensuing discussion, it is assumed without loss of generality that S is a sequence of integers, as in the preceding example. Selection will therefore call for finding the kth smallest element. We also introduce the following useful notation. For a real number r, $\lfloor r \rfloor$ denotes the largest integer smaller than or equal to r (the "floor" of r), while $\lceil r \rceil$ denotes the smallest integer larger than or equal to r (the "ceiling" of r). Thus $\lfloor 3.9 \rfloor = 3$, $\lceil 3.1 \rceil = 4$, and $\lfloor 3.0 \rfloor = \lceil 3.0 \rceil = 3$.

2.2.4 Complexity

Three particular values of k in the definition of the selection problem immediately come to one's mind: $k = 1$, $k = n$, and $k = \lceil n/2 \rceil$. In the first two cases we would be looking for the smallest and largest elements of S, respectively. In the third case, $s_{(k)}$ would be the median of S, that is, the element for which half of the elements of S are smaller than (or equal to) it and the other half larger (or equal). It seems intuitive, at least in the sequential mode of thinking and computing, that the first two cases are easier to solve than when $k = \lceil n/2 \rceil$ or any other value. Indeed, for $k = 1$ or $k = n$, all one has to do is examine the sequence element by element, keeping track of the smallest (or largest) element seen so far until the result is obtained. No such obvious solution appears to work for $1 < k < n$.

Evidently, if S were presented in sorted order, that is, $S = \{s_{(1)}, s_{(2)}, \ldots, s_{(n)}\}$, then selection would be trivial: In one step we could obtain $s_{(k)}$. Of course, we do not assume that this is the case. Nor do we want to sort S first and then pick the kth element: This appears to be (and indeed is) a computationally far more demanding task than we need (particularly for large values of n) since sorting would solve the selection problem for all values of k, not just one.

Regardless of the value of k, one fact is certain: In order to determine the kth

smallest element, we must examine each element of **S** at least once. This establishes a lower bound of $\Omega(n)$ on the number of (sequential) steps required to solve the problem. From chapter 1, we know that this immediately implies an $\Omega(n)$ lower bound on the cost of any parallel algorithm for selection.

2.3 A SEQUENTIAL ALGORITHM

In this section we study a sequential algorithm for the selection problem. There are two reasons for our interest in a sequential algorithm. First, our parallel algorithm is based on the sequential one and is a parallel implementation of it on an EREW SM SIMD computer. Second, the parallel algorithm assumes the existence of the sequential one and uses it as a procedure.

The algorithm presented in what follows in the form of procedure SEQUENTIAL SELECT is recursive in nature. It uses the divide-and-conquer approach to algorithm design. The sequence **S** and the integer k are the procedure's initial input. At each stage of the recursion, a number of elements of **S** are discarded from further consideration as candidates for being the kth smallest element. This continues until the kth element is finally determined. We denote by $|S|$ the size of a sequence **S;** thus initially, $|S| = n$. Also, let Q be a small integer constant to be determined later when analyzing the running time of the algorithm.

PRELIMINARY RESULTS

procedure SEQUENTIAL SELECT (S, k)

Step 1: **if** $|S| < Q$ then sort S and return the kth element directly
else subdivide S into $|S|/Q$ subsequences of Q elements each (with up to $Q-1$ leftover elements)
end **if.**

Step 2: Sort each subsequence and determine its median.

Step 3: Call SEQUENTIAL SELECT recursively to find m, the median of the $|S|/Q$ medians found in step 2.

Step 4: Create three subsequences S_1, S_2, and S_3 of elements of S smaller than, equal to, and larger than m, respectively.

Step 5: if $|S_1| \geq k$ then (the kth element of S must be in S_1)
call SEQUENTIAL SELECT recursively to find the kth element of S_1
else **if** $|S_1| + |S_2| \geq k$ then return m
else call SEQUENTIAL SELECT recursively to find the $(k - |S_1| - |S_2|)$th element of S_3
end **if**
end **if.** □

Note that the preceding statement of procedure SEQUENTIAL SELECT does not specify how the kth smallest element of S is actually returned. One way to do this would be to have an additional parameter, say, **x,** in the procedure's heading (besides

S and k) and return the kth smallest element in **x.** Another way would be to simply return the kth smallest as the first element of the sequence *S*.

Analysis. A step-by-step analysis of $t(n)$, the running time of SEQUENTIAL SELECT, is now provided.

Step 1: Since **Q** is a constant, sorting **S** when $|S| < \mathbf{Q}$ takes constant time. Otherwise, subdividing **S** requires $c_1 n$ time for some constant c_1.

Step **2:** Since each of the $|S|/Q$ subsequences consists of **Q** elements, it can be sorted in constant time. Thus, $c_2 n$ time is also needed for this step for some constant c_2.

Step **3:** There are $|S|/Q$ medians; hence the recursion takes $t(n/Q)$ time.

Step **4:** One pass through **S** creates $\mathbf{S}_1$, S_2, and S_3 given m; therefore this step is completed in $c_3 n$ time for some constant c_3.

Step 5: Since m is the median of $|S|/Q$ elements, there are $|S|/2Q$ elements larger than or equal to it, as shown in Fig. **2.1.** Each of the $|S|/Q$ elements was itself the median of a set of **Q** elements, which means that it has $Q/2$ elements larger than or equal to it. It follows that $(|S|/2Q) \times (Q/2) = |S|/4$ elements of **S** are guaranteed to be larger than or equal to m. Consequently, $|S_1| \leqslant 3|S|/4$. By a similar reasoning, $|S_3| \leqslant 3|S|/4$. A recursive call in this step to SEQUENTIAL SELECT therefore requires $t(3n/4)$. From the preceding analysis we have

$$t(n) = c_4 n + t(n/Q) + t(3n/4), \quad \text{where } c_4 = c_1 + c_2 + c_3.$$

The time has now come to specify **Q.** If we choose **Q** so that

$$n/Q + 3n/4 < n,$$

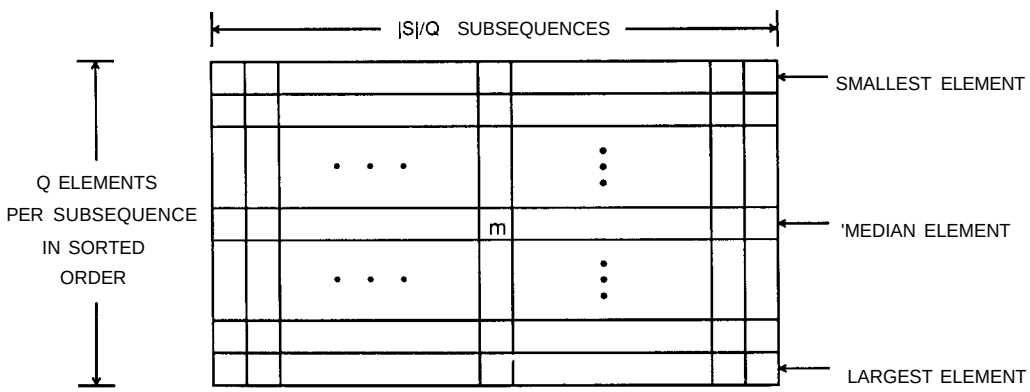


Figure 2.1 Main idea behind procedure SEQUENTIAL SELECT.

then the two recursive calls in the procedure are performed on ever-decreasing sequences. Any value of $Q \geqslant 5$ will do. Take $Q = 5$; thus

$$t(n) = c_4 n + t(n/5) + t(3n/4).$$

This recurrence can be solved by assuming that

$$t(n) \leqslant c_5 n \quad \text{for some constant } c_5.$$

Substituting, we get

$$\begin{aligned} t(n) &\leqslant c_4 n + c_5(n/5) + c_5(3n/4) \\ &= c_4 n + c_5(19n/20). \end{aligned}$$

Finally, taking $c_5 = 20c_4$ yields

$$\begin{aligned} t(n) &\leqslant c_5(n/20) + c_5(19n/20) \\ &= c_5 n, \end{aligned}$$

thus confirming our assumption. In other words, $t(n) = O(n)$, which is optimal in view of the lower bound derived in section 2.2.4.

2.4 DESIRABLE PROPERTIES FOR PARALLEL ALGORITHMS

Before we embark in our study of a parallel algorithm for the selection problem, it may be worthwhile to set ourselves some design goals. A number of criteria were described in section 1.3 for evaluating parallel algorithms. In light of these criteria, five important properties that we desire a parallel algorithm to possess are now defined.

2.4.1 Number of Processors

The first two properties concern the number of processors to be used by the algorithm. Let n be the size of the problem to be solved:

(i) ***p(n) must be smaller than n:*** No matter how inexpensive computers become, it is unrealistic when designing a parallel algorithm to assume that we have at our disposal more (or even as many) processors as there are items of data. This is particularly true when n is very large. It is therefore important that $p(n)$ be expressible as a sublinear function of n, that is, $p(n) = n^x$, $0 < x < 1$.

(ii) ***p(n) must be adaptive:*** In computing in general, and in parallel computing in particular, "appetite comes with eating." The availability of additional computing power always means that larger and more complex problems will be attacked than was possible before. Users of parallel computers will want to push their machines to their limits and beyond. Even if one could afford to have as many processors as data for a particular problem size, it may not be desirable to design an algorithm based on that assumption: A larger problem would render the algorithm totally useless.

Algorithms using a number of processors that is a sublinear function of n [and hence satisfying property (i)], such as $\log n$ or $n^{1/2}$, would not be acceptable either due to their inflexibility. What we need are algorithms that possess the "intelligence" to adapt to the actual number of processors available on the computer being used.

2.4.2 Running Time

The next two properties concern the worst-case running time of the parallel algorithm:

(i) ***$t(n)$ must be small:*** Our primary motive for building parallel computers is to speed up the computation process. It is therefore important that the parallel algorithms we design be fast. To be useful, a parallel algorithm should be significantly faster than the best sequential algorithm for the problem at hand.

(ii) ***$t(n)$ must be adaptive:*** Ideally, one hopes to have an algorithm whose running time decreases as more processors are used. In practice, it is usually the case that a limit is eventually reached beyond which no speedup is possible regardless of the number of processors used. Nevertheless, it is desirable that $t(n)$ vary inversely with $p(n)$ within the bounds set for $p(n)$.

2.4.3 Cost

Ultimately, we wish to have parallel algorithms for which $c(n) = p(n)\ x\ t(n)$ always matches a known lower bound on the number of sequential operations required in the worst case to solve the problem. In other words, a parallel algorithm should be *cost optimal.*

In subsequent chapters we shall see that meeting the preceding objectives is usually difficult and sometimes impossible. In particular, when a set of processors are linked by an interconnection network, the geometry of the network often imposes limits on what can be accomplished by a parallel algorithm. It is a different story when the algorithm is to run on a shared-memory parallel computer. Here, it is not at all unreasonable to insist on these properties given how powerful and flexible the model is.

In section 2.6 we describe a parallel algorithm for selecting the kth smallest element of a sequence $S = \{s_1, s_2, \ldots, s_n\}$. The algorithm runs on an EREW SM SIMD computer with N processors, where $N < n$. The algorithm enjoys all the desirable properties formulated in this section:

(i) It uses $p(n) = n^{1-x}$ processors, where $0 < x < 1$. The value of x is obtained from $N = n^{1-x}$. Thus $p(n)$ is sublinear and adaptive.

(ii) It runs in $t(n) = O(n^x)$ time, where x depends on the number of processors available on the parallel computer. The value of x is obtained in (i). Thus $t(n)$ is smaller than the running time of the optimal sequential algorithm described in

section 2.3. It is also adaptive: The larger is $p(n)$, the smaller is $t(n)$, and vice versa.

(iii) It has a cost of $c(n) = n^{1-x} \times O(n^x) = O(n)$, which is optimal in view of the lower bound derived in section 2.2.4.

In closing this section we note that all real quantities of the kind just described (e.g., n^{1-x} and n^x) should in practice be rounded to a convenient integer, according to our assumption in chapter 1. When dealing with numbers of processors and running times, though, it is important that this rounding be done pessimistically. Thus, the real n^{1-x} representing the number of processors used by an algorithm should be interpreted as $\lfloor n^{1-x} \rfloor$: This is to ensure that the resulting integer does not exceed the actual number of processors. Conversely, the real n^x representing the worst-case running time of an algorithm should be interpreted as $\lceil n^x \rceil$: This guarantees that the resulting integer is not smaller than the true worst-case running time.

2.5 TWO USEFUL PROCEDURES

In the EREW SM SIMD model no two processors can gain access to the same memory location simultaneously. However, two situations may arise in a typical parallel algorithm:

(i) All processors need to read a datum held in a particular location of the common memory.

(ii) Each processor has to compute a function of data held by other processors and therefore needs to receive these data.

Clearly, a way must be found to efficiently simulate these two operations that cannot be performed in one step on the EREW model. In this section, we present two procedures for performing these simulations. The two procedures are used by the algorithm in this chapter as well as by other parallel algorithms to be studied subsequently. In what follows we assume that N processors P_1, P,, ..., P_N are available on an EREW SM SIMD computer.

2.5.1 Broadcasting a Datum

Let D be a location in memory holding a datum that all N processors need at a given moment during the execution of an algorithm. As mentioned in section 1.2.3.1, this is a special case of the more general multiple-read situation and can be simulated on an EREW computer by the broadcasting process described in example 1.4. We now give this process formally as procedure BROADCAST. The procedure assumes the presence of an array A of length N in memory. The array is initially empty and is

used by the procedure as a working space to distribute the contents of D to the processors. Its ith position is denoted by $A(i)$.

procedure BROADCAST (D, N, A)

Step 1: Processor P_1
(i) reads the value in D,
(ii) stores it in its own memory, and
(iii) writes it in $A(1)$.

Step 2: **for** $i=0$ **to** (log $\mathbf{N}-1$) **do**
for $j=2^i+1$ **to** 2^{i+1} **do in parallel**
Processor P_j
(i) reads the value in $A(j-2^i)$,
(ii) stores it in its own memory, and
(iii) writes it in $A(j)$.
end for
end for. □

PRELIMINARY RESULTS

The working of BROADCAST is illustrated in Fig. 2.2 for N = 8 and D = 5. When the procedure terminates, all processors have stored the value of D in their local memories for later use. Since the number of processors having read D doubles in each iteration, the procedure terminates in $O(\log N)$ time. The memory requirement of BROADCAST is an array of length N. Strictly speaking, an array of half that length will do since in the last iteration of the procedure all the processors have received the value in D and need not write it back in A [see Fig. 2.2(d)]. BROADCAST can be easily modified to prevent this final write operation and hence use an array A of length $N/2$.

Besides being generally useful in broadcasting data to processors during the execution of an algorithm, procedure BROADCAST becomes particularly important when starting an adaptive algorithm such as the one to be described in section 2.6. Initially, each of the N processors knows its own index i, $1 \leqslant i \leqslant N$, and the available number of processors N. When a problem is to be solved, the problem size n must be communicated to all processors. This can be done using procedure BROADCAST before executing the algorithm. Each processor now computes x from $N = n^{1-x}$, and the algorithm is performed. Therefore, we shall assume henceforth that the parameter **x** is known to all processors when an adaptive algorithm starts its computation.

2.5.2 Computing All Sums

Assume that each processor P_i holds in its local memory a number a,, $1 \leqslant i \leqslant N$. It is often useful to compute, for each P_i, the sum a, + a, + ... + a,. In example 1.5 an algorithm was demonstrated for computing the sum of N numbers in $O(\log N)$ time on a tree-connected computer with $O(N)$ processors. Clearly this algorithm can be implemented on a shared-memory machine to compute the sum in the same amount of time using the same number of processors. The question here is: Can the power

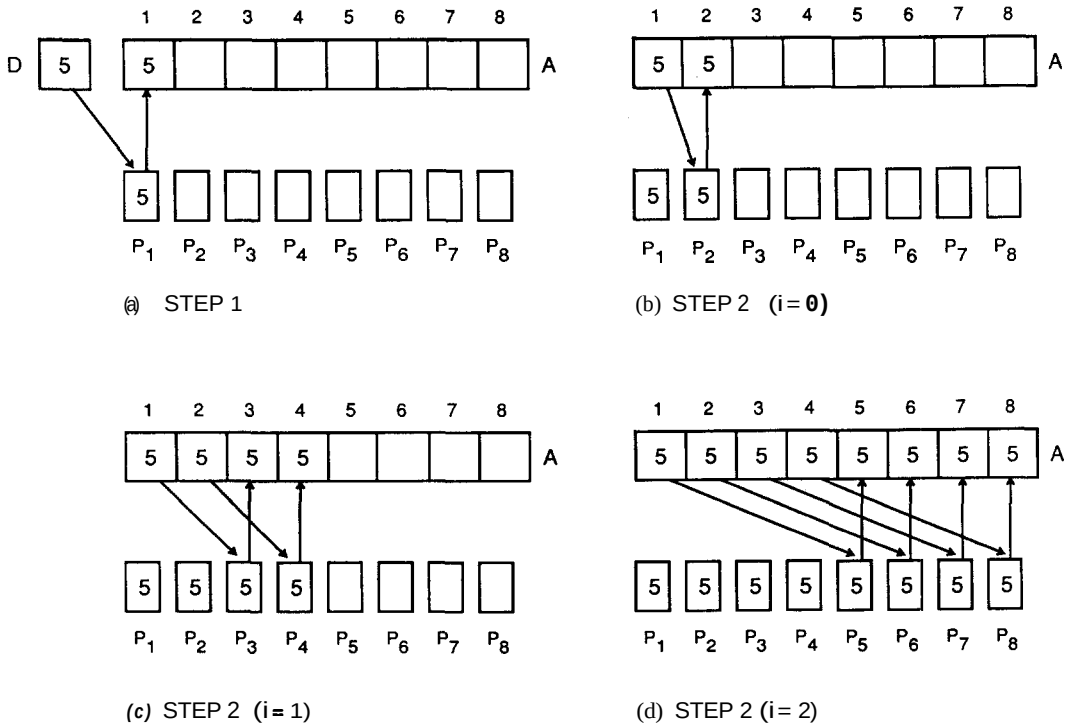


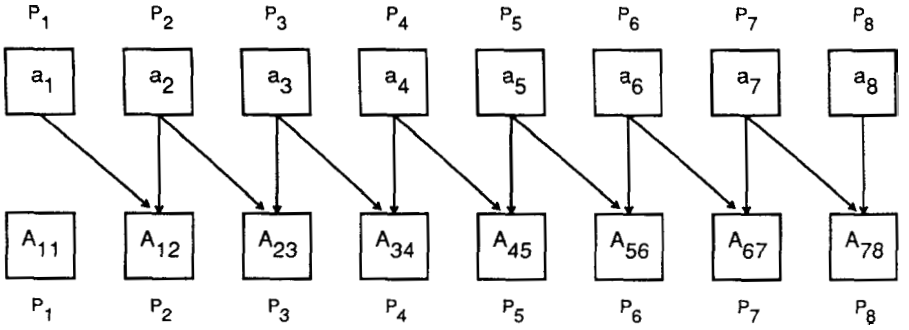
Figure 2.2 Distributing a datum to eight processors using procedure BROADCAST.

of the shared-memory model be exploited to compute ***all*** sums of the form $a_1 + a_2 + \cdots + a_i$, $1 \leqslant i \leqslant N$, known as the *prefix* ***sums,*** using N processors in $O(\log N)$ time? As it turns out, this is indeed possible. The idea is to keep as many processors busy as long as possible and exploit the associativity of the addition operation. Procedure **ALLSUMS** given formally in the following accomplishes exactly that:

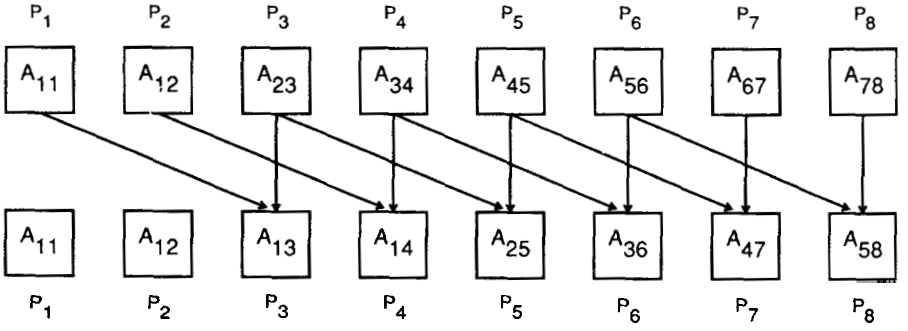
PRELIMINARY RESULTS

```
procedure ALLSUMS (a_1, a_2, ..., a_n)
  for j = 0 to log N − 1 do
    for i = 2^j + 1 to N do in parallel
        Processor P_i
          (i) obtains a_{i−2^j} from P_{i−2^j} through shared memory and
          (ii) replaces a_i with a_{i−2^j} + a_i.
    end for
  end for. □
```

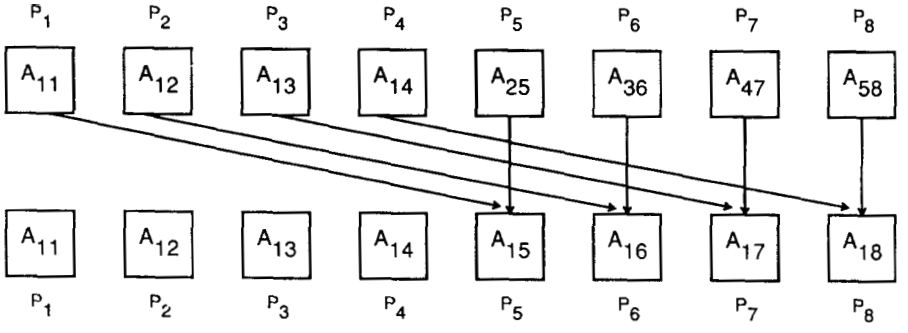
The working of **ALLSUMS** is illustrated in Fig. 2.3 for N = 8 with A_{ij} referring to the sum $a_i + a_{i+1} + \ldots + a_j$. When the procedure terminates, a_i has been replaced by



(a) j = 0



(b) j = 1



(c) j = 2

Figure 2.3 Computing the prefix sums of eight numbers using procedure ALLSUMS

$a_1 + a_2 + \ldots + a_i$ in the local memory of P_i, for $1 \leqslant i \leqslant N$. The procedure requires $O(\log N)$ time since the number of processors that have finished their computation doubles at each stage.

It is important to note that procedure **ALLSUMS** can be modified to solve any problem where the addition operation is replaced by any other associative binary operation. Examples of such operations on numbers are multiplication, finding the larger or smaller of two numbers, and so on. Other operations that apply to a pair of logical quantities (or a pair of bits) are **and,** or, and xor. Various aspects of the problem of computing the prefix sums in parallel are discussed in detail in chapters 13 and 14.

2.6 AN ALGORITHM FOR PARALLEL SELECTION

PRELIMINARY RESULTS

We are now ready to study an algorithm for parallel selection on an EREW SM SIMD computer. The algorithm presented as procedure PARALLEL SELECT makes the following assumptions (some of these were stated earlier):

1. A sequence of integers $S = \{s_1, s_2, \ldots, s_n\}$ and an integer k, $1 \leqslant k \leqslant n$, are given, and it is required to determine the kth smallest element of S. This is the initial input to PARALLEL SELECT.
2. The parallel computer consists of N processors $P_1, P_2, \ldots, P_N$.
3. Each processor has received n and computed x from $N = n^{1-x}$, where $0 < x < 1$.
4. Each of the n^{1-x} processors is capable of storing a sequence of n^x elements in its local memory.
5. Each processor can execute procedures SEQUENTIAL SELECT, BROADCAST, and ALLSUMS.
6. M is an array in shared memory of length N whose ith position is $M(i)$.

procedure PARALLEL SELECT (S, k)

Step 1: **if** $|S| \leqslant 4$ **then** P_1 uses at most five comparisons to return the kth element
else
(i) S is subdivided into $|S|^{1-x}$ subsequences S_i of length $|S|^x$ each, where $1 \leqslant i \leqslant |S|^{1-x}$, and
(ii) subsequence S_i is assigned to processor P_i.
end if.

Step 2: **for** $i = 1$ **to** $|S|^{1-x}$ **do in parallel**
(2.1) {P_i obtains the median m_i, i.e., the $\lceil |S_i|/2 \rceil$th element, of its associated subsequence)
SEQUENTIAL SELECT $(S_i, \lceil |S_i|/2 \rceil)$
(2.2) P_i stores m_i in $M(i)$
end for.

Step 3: {The procedure is called recursively to obtain the median m of M}

PARALLEL SELECT $(M, \lceil |M|/2 \rceil)$.

Step 4: The sequence S is subdivided into three subsequences:

$L = \{s_i \in S: s_i < m\}$,
$E = \{s_i \in S: s_i = m\}$, and
$G = \{s_i \in S: s_i > m\}$.

Step 5: **if** $|L| \geq k$ **then** PARALLEL SELECT (L, k)
else **if** $|L| + |E| \geq k$ **then** return **m**
else PARALLEL SELECT $(G, k - |L| - |E|)$
end if
end if. □

Note that the precise mechanism used by procedure PARALLEL SELECT to return the kth smallest element of S is unspecified in the preceding statement. However, any of the ways suggested in section 2.3 in connection with procedure SEQUENTIAL SELECT can be used here.

Analysis. We have deliberately given a high-level description of PARALLEL SELECT to avoid obscuring the main ideas of the algorithm. In order to obtain an accurate analysis of the procedure's running time, however, various implementation details must be specified. As usual, we denote by $t(n)$ the time required by PARALLEL SELECT for an input of size n. A function describing $t(n)$ is now obtained by analyzing each step of the procedure.

Step 1: To perform this step, each processor needs the beginning address A of sequence S in the shared memory, its size $|S|$, and the value of k. These quantities can be broadcast to all processors using procedure BROADCAST: This requires $O(\log n^{1-x})$ time. If $|S| \leq 4$, then P, returns the kth element in constant time. Otherwise, P_i computes the address of the first and last elements in S_i from $A + (i - 1)n^x$ and $A + in^x - 1$, respectively; this can be done in constant time. Thus, step 1 takes $c_1 \log n$ time units for some constant c_1.

Step 2: SEQUENTIAL SELECT finds the median of a sequence of length n^x in $c_2 n^x$ time units for some constant c_2.

Step 3: Since PARALLEL SELECT is called with a sequence of length n^{1-x}, this step requires $t(n^{1-x})$ time.

Step 4: The sequence S can be subdivided into L, E, and G as follows:

(i) First **m** is broadcast to all the processors in $O(\log n^{1-x})$ time using procedure BROADCAST.

(ii) Each processor P_i now splits S_i into three subsequences L_i, E_i, and G_i of elements smaller than, equal to, and larger than m, respectively. This can be done in time linear in the size of S_i, that is, $O(n^x)$ time.

(iii) The subsequences L_i, E_i, and G_i are now merged to form L, E, and G. We show how this can be done for the L_i; similar procedures with the same

running time can be derived for merging the E_i and G_i, respectively. Let $a_i = |L_i|$. For each i, $1 \leqslant i \leqslant n^{1-x}$, the sum

$$z_i = \sum_{j=1}^{i} a_j$$

is computed. All these sums can be obtained by n^{1-x} processors in $O(\log n^{1-x})$ time using procedure ALLSUMS. Now let $z_0 = 0$. All processors simultaneously merge their L_i subsequences to form L; Processor P_i copies L_i into L starting at position $z_{i-1} + 1$. This can be done in $O(n^x)$ time.

Hence the time required by this step is $c_3 n^x$ for some constant c_3.

Step 5: The size of L needed in this step has already been obtained in step 4 through the computation of $z_{n^{1-x}}$. The same remark applies to the sizes of E and G. Now we must determine how much time is taken by each of the two recursive steps. Since m is the median of M, $n^{1-x}/2$ elements of S are guaranteed to be larger than it. Furthermore, every element of M is smaller than at least $n^x/2$ elements of S. Thus $|L| \leqslant 3n/4$. Similarly, $|G| \leqslant 3n/4$. Consequently, step 5 requires at most $t(3n/4)$ time.

The preceding analysis yields the following recurrence for $t(n)$:

$$t(n) = c_1 \log n + c_2 n^x + t(n^{1-x}) + c_3 n^x + t(3n/4),$$

whose solution is $t(n) = O(n^x)$ for $n > 4$. Since $p(n) = n^{1-x}$, we have

$$c(n) = p(n) \times t(n) = n^{1-x} \times O(n^x) = O(n).$$

This cost is optimal in view of the $\Omega(n)$ lower bound derived in section 2.2. Note, however, that n^x is asymptotically larger than log n for any x. (Indeed we have used this fact in our analysis of PARALLEL SELECT.) Since $N = n^{1-x}$ and $n/n^x < n/\log n$, it follows that PARALLEL SELECT is cost optimal provided $N < n/\log n$.

Example 21

This example illustrates the working of PARALLEL SELECT. Let S = {3, 14, 16, 20, 8, 31, 22, 12, 33, 1, 4, 9, 10, 5, 13, 7, 24, 2, 14, 26, 18, 34, 36, 25, 14, 27, 32, 35, 33}, that is, n = 29 and let k = 21, that is, we need to determine the twenty-first element of S. Assume further that the EREW SM SIMD computer available consists of five processors, (N = 5). Hence $|S|^{1-x} = 5$, implying that $1 - x = 0.47796$. The input sequence is initially in the shared memory as shown in Fig. 2.4(a). After step 1, each processor has been assigned a subsequence of S: The first four processors receive six elements each, and the fifth receives five, as in Fig. 2.4(b). Now each processor finds the median of its subsequence in step 2 and places it in a shared-memory array M; this is illustrated in Fig. 2.4(c). When PARALLEL SELECT is called recursively in step 3, it returns the median $m = 14$ of M. The three subsequences of S, namely, L, E, and G of elements smaller than, equal to, and larger than 14, respectively, are formed in step 4, as shown in Fig. 2.4(d). Since $|L| = 11$ and $|E| = 3$, $|L| + |E| < k$ and PARALLEL SELECT is called recursively in step 5 with S = G and $k = 21 - (11 + 3) = 7$. Since $|G| = 15$, we use $15^{1-x} = 3.6485$, that is, three, processors during this recursive step.

Again in step 1, each processor is assigned five elements, as shown in Fig. 2.4(e). The sequence M of medians obtained in step 2 is shown in Fig. 2.4(f). The median $m = 26$

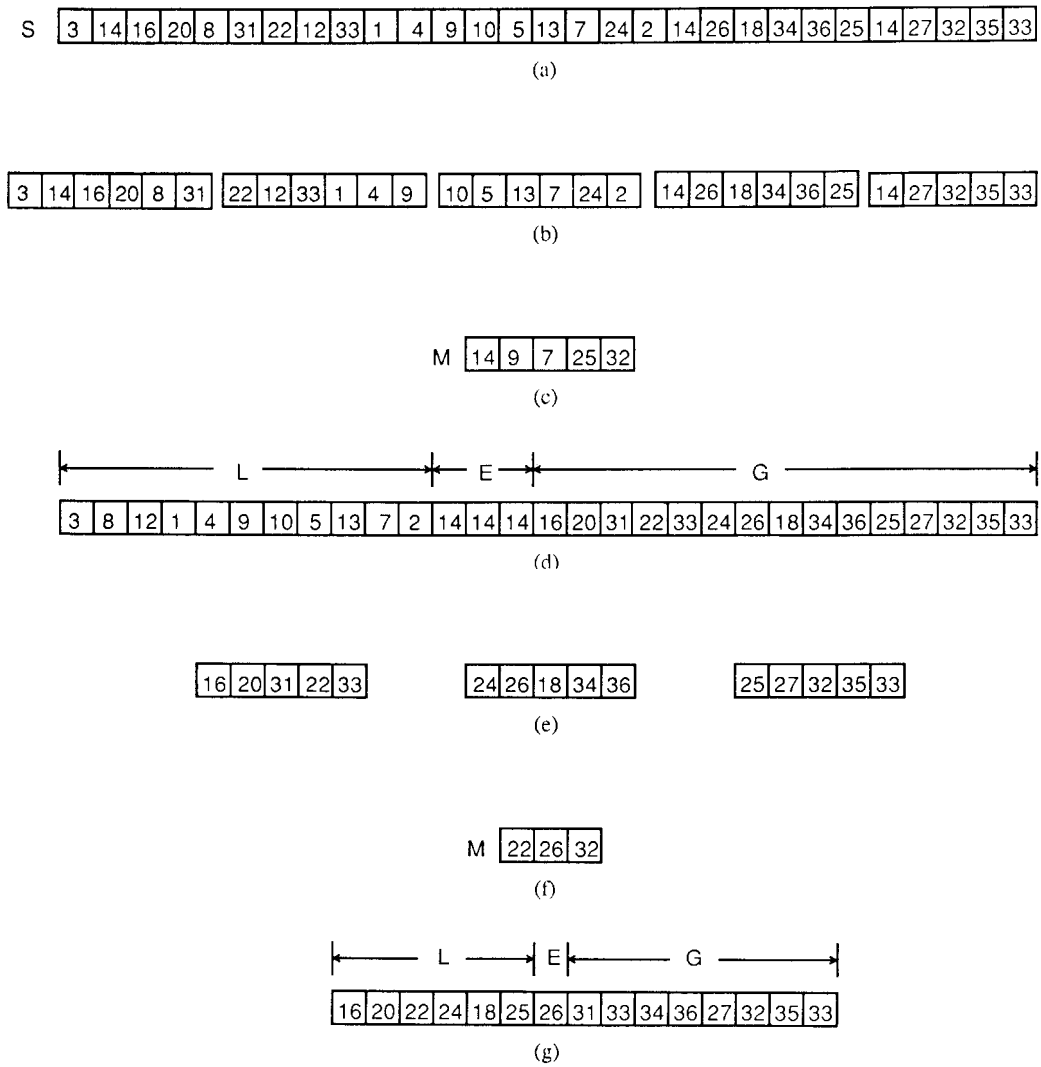


Figure 2.4 Selecting twenty-first element of a sequence using procedure PARALLEL SELECT.

of M is determined in step 3. The three subsequences *L*, *E*, and *G* created in step 4 are illustrated in Fig. 2.4(g). Since $|L| = 6$ and $|E| = 1$, the only element of E, namely, 26, is returned as the twenty-first element of the input. □

We conclude this section with the following observation. In designing PARALLEL SELECT, we adopted the approach of taking a sequential algorithm for a problem and turning it into a parallel algorithm. We were quite successful in

obtaining an algorithm for the EREW SM SIMD model that is fast, adaptive, and cost optimal while using a number of processors that is sublinear in the size of the input. There are problems however, for which this approach does not work that well. In these cases a parallel algorithm (not based on any sequential algorithm) must be derived by exploiting the inherent parallelism in the problem. We shall study such algorithms in subsequent chapters. Taken to the extreme, this latter approach can sometimes offer surprises: A parallel algorithm provides an insight that leads to an improvement over the best existing sequential algorithm.

2.7 PROBLEMS

2.1 In an interconnection-network SIMD computer, one of the N processors holds a datum that it wishes to make known to all other processors. Show how this can be done on each of the networks studied in chapter 1. Which of these networks accomplish this task in the same order of time as required by procedure BROADCAST?

2.2 Consider an SIMD computer where the N processors are linked together by a perfect shuffle interconnection network. Now assume that the line connecting two processors can serve as a two-way link; in other words, if P_i can send data to P_j (using a perfect shuffle link), then P_j can also send data back to P_i (the latter link being referred to as a perfect *unshuffle* connection). In addition, assume that for $i < N - 1$, each P_i is linked by a direct one-way link to P_{i+1}; call these the nearest-neighbor links. Each processor P_i holds an integer a_i. It is desired that a_i in P_i be replaced with $a_1 + a_2 + \ldots + a_i$ for all i. Can this task be accomplished using the unshuffle and nearest-neighbor links in the same order of time as required by procedure ALLSUMS?

2.3 A parallel selection algorithm that uses $O(n/\log^s n)$ processors and runs in $O(\log^s n)$ time for some $0 \leq s \leq 1$ would be faster than PARALLEL SELECT since $\log^s n$ is asymptotically smaller than n^x for any x and s. Can you find such an algorithm?

2.4 If PARALLEL SELECT were to be implemented on a CREW SM SIMD computer, would it run any faster?

2.5 Design and analyze a parallel algorithm for solving the selection problem on a CRCW SM SIMD computer.

2.6 A tree-connected computer with n leaves stores one integer of a sequence S per leaf. For a given k, $1 \leq k \leq n$, design an algorithm that runs on this computer and selects the kth smallest element of S.

2.7 Repeat problem 2.6 for a linear array of n processors with one element of S per processor.

2.8 Repeat problem 2.6 for an $n^{1/2} \times n^{1/2}$ mesh of processors with one element of S per processor.

2.9 Consider the following variant of the linear array interconnection network for SIMD computers. In addition to the usual links connecting the processors, a further communication path known as a bus is available, as shown in Fig. 2.5. At any given time during the execution of an algorithm, precisely one of the processors is allowed to broadcast one of the input data to the other processors using the bus. All processors receive the datum simultaneously. The time required by the broadcast operation is assumed to be constant. Repeat problem 2.6 for this modified linear array.

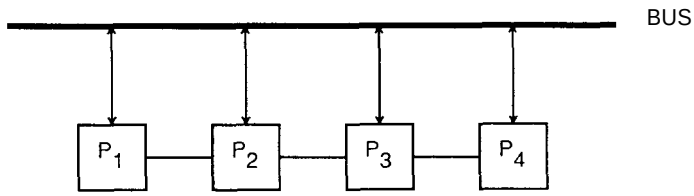


Figure 2.5 Linear array with a bus.

2.10 Modify the mesh interconnection network for SIMD machines to include a bus and repeat problem 2.6 for the modified model.

2.11 Design an algorithm for solving the selection problem for the case $k = 1$ (i.e., finding the smallest element of a sequence) on each of the following two models: (i) a mesh-connected SIMD computer and (ii) the machine in problem 2.10.

2.12 A problem related to selection is that of determining the k smallest elements of a sequence S (in any order). On a sequential computer this can be done as follows: First determine the kth smallest element (using SEQUENTIAL SELECT); then one pass through S suffices to determine the $k - 1$ elements smaller than k. The running time of this algorithm is linear in the size of S. Design a parallel algorithm to solve this problem on your chosen submodel of each of the following models and analyze its running time and cost: (i) shared-memory SIMD, (ii) interconnection-network SIMD, and (iii) specialized architecture.

2.13 Modify procedure BROADCAST to obtain a formal statement of procedure STORE described in section 1.2.3.1. Provide a different version of your procedure for each of the write conflict resolution policies mentioned in chapter 1.

2.14 In steps 1 and 2 of procedure SEQUENTIAL SELECT, a simple sequential algorithm is required for sorting short sequences. Describe one such algorithm.

2.8 BIBLIOGRAPHICAL REMARKS

As mentioned in section 2.1, the problem of selection has a number of applications in computer science and statistics. In this book, for example, we invoke a procedure for selecting the kth smallest out of n elements in our development of algorithms for parallel merging (chapter 3), sorting (chapter 4), and convex huil computation (chapter 11). An application to image analysis is cited in [Chandran]. In statistics, selection is referred to as the computation of order *statistics.* In particular, computing the median element of a set of data is a standard procedure in statistical analysis. The idea upon which procedure SEQUENTIAL SELECT is based was first proposed in [Blum]. Sequential algorithms for sorting short sequences, as required by that procedure, can be found in [Knuth].

Procedures BROADCAST and ALLSUMS are adapted from [Akl 2]. Another way of computing the prefix sums of n numbers is through a specialized network of processors. One such network is suggested by Fig. 2.3. It consists of $\log n$ rows of n processors each. The processors are connected by the lines illustrating the flow of data in Fig. 2.3. The top row of processors receives the n numbers as input, and all the prefix sums are produced by the bottom