Paralelní třídění

Tomáš Oberhuber
tomas.oberhuber@fjfi.cvut.cz

15. dubna 2024

Videa na Youtube:

Bubble sort a quicksort

Radix sort a bitonic sort

Třídící algoritmy

Vstup: Posloupnost prvků $a_1, \ldots, a_n$ s definovaným uspořádáním $\leq$.
Výstup: Setříděná posloupnost, pro kterou platí $a_{i_1} \leq a_{i_2} \ldots a_{i_n}$.

- rozlišujeme vnitřní a vnější třídění
  - my se budeme zabývat jen vnitřním

Výstupní sekvence může být uložena:

- v paměti jednoho výpočetního uzlu
- distribuovaně
  - požadujeme pak, aby pro $i < j$ platilo, že všechny prvky uložené na uzlu P_i byly menší než všechny prvky na uzlu P_j

Bublinkové třídění

Sekvenční algoritmus:

- porovnáváme a uspořádáváme postupně prvky:

$$(a_1, a_2), (a_2, a_3), \ldots (a_{n-1}, a_n)$$

- tím se největší prvek dostane na poslední pozici
- tuto iteraci opakuji ještě $n - 1$ krát
- $T_S(n) = \sum_{i=1}^{n} i = \frac{n(n-1)}{2}$
- v této podobě nelze algoritmus paralelizovat

Paralelní bublinkové třídění

- budeme střídavě provádět porovnávání sudých a lichých dvojic tj.

$$(a_1, a_2), (a_3, a_4), \ldots (a_{n-1}, a_n)$$

$$(a_2, a_3), (a_4, a_5), \ldots (a_{n-2}, a_{n-1})$$

- nazývá se také **odd-even sort**
- nyní již lze provádět porovnání a uspořádání v rámci jednoho kroku paralelně
- každý prvek se může v jednom kroku posunout maximálně o dvě místa
- postup tedy musím opakovat $\frac{n}{2}$-krát, pokaždé dělám $n - 1$ porovnání
- $T_S(n) = \frac{n(n-1)}{2}$

Paralelní bublinkové třídění - analýza

- $T_P(n,p) = \frac{n}{2}\frac{n-1}{p} = \theta(\frac{n^2}{2p})$
- $S(n,p) = \frac{n^2}{2}\frac{2p}{n^2} = p$
- $E(n,p) = 1$
- $C(n,p) = \frac{n^2}{2} = \theta(T_S(n^2))$
- p může být maximálně $\frac{n}{2}$, pak je $T_P(n,p) = n$
- to není moc dobrý výsledek v porovnání s $n \log n$ u quicksortu, uvážíme-li, že jsme zaměstnali $\frac{n}{2}$ procesorů

Paralelní bublinkové třídění - implementace

- implementace na architekturách se sdílenou pamětí
  - jednotlivá porovnání tvoří prvotní tasky
  - sloučím je do bloků a ty pak mapuji na jednotlivé procesy
  - tím zabráním, aby různé procesy sahaly na blízké prvky v poli
- implementace na architekturách s distribuovanou pamětí
  - pokud rozdistribuuji vstupní posloupnost např. na dva uzly takto
    - $a_1, a_2, \ldots a_{\frac{n}{2}-1}$ a $a_{\frac{n}{2}}, \ldots a_n$
  - pak porovnání $(a_{\frac{n}{2}-1}, a_{\frac{n}{2}})$ se účastní dva různé procesy
  - porovnání probíhá takto
    - oba procesy si vzájemně vymění svá čísla, tj. oba budou mít $a_{\frac{n}{2}-1}$ i $a_{\frac{n}{2}}$
    - proces s vyšším ID si nechá větší z obou čísel, proces s nižším ID si nechá to menší

Paralelní bublinkové třídění - implementace

- implementace v CUDA probíhá takto
  - každý blok si načte $2 * blockDim.x$ prvků do sdílené paměti
  - každé vlákno pak střídavě provádí odd-even sort v rámci bloku
  - až se setřídí celý blok, provede se zápis do globální paměti
  - pak se vše opakuje, ale bloky si načtou data s posunutím rovným $blockDim.x$, tj polovina prvků tříděných blokem
  - po setřídění v rámcí bloků a zapsání výsledku do globální paměti

Quicksort

- jde o algoritmus založený na metodě rozděl a panuj
- je známý svou efektivitou se složitostí $T_S(n) = \theta(n \log n)$
- autorem je C. A. R. Hoare, 1962
- algoritmus pracuje tak, že v každém kroku rozdělí posloupnost

  $$a_1, \ldots, a_m$$

  na dvě

  $$a_{i_1}, \ldots a_{i_l} \text{ a } a_{i_l+1}, \ldots a_{i_m},$$

  tak, že každý prvek z první posloupnosti je menší než všechny prvky z druhé posloupnosti
- toto rozdělení určuje předem zvolený pivot
- obě podposloupnosti se zpracují rekurzivně stejným způsobem

Quicksort sekvenčně

```
 1  void quickSort( int* data, const int first, const int last )
 2  {
 3     int pivot = data[ last ];
 4     int i( first - 1 ), j( last );
 5     while( true )
 6     {
 7        while( data[ ++i ] < pivot );
 8        while( data[ --j ] > pivot )
 9           if( j == first )
10              break;
11        if( i >= j )
12           break;
13        swap( data[ i ], data[ j ] );
14     }
15     swap( data[ i ], data[ last ] );
16     if( first < i - 1 )
17        quickSort( data, first, i - 1 );
18     if( i + 1 < last )
19        quickSort( data, i + 1, last );
20  }
```

Quicksort paralelně

- paralelizaci lze provést pomocí datové dekompozice
- zpracování podposloupností můžeme provádět nezávisle a tedy paralelně
  - na začátku ale máme málo paralelismu
  - tento přístup také není vhodný pro architektury s distribuovanou pamětí
- paralelní quicksortu je mnohem citlivější na volbu pivota
  - špatně zvolený pivot nejen snižuje efektivitu algoritmu, ale také vede k špatně vybalancované zátěži jednotlivých procesorů

Quicksort pro architektury se sdílenou pamětí

- předpokládáme tedy, že nesetříděná posloupnost je rozdistribuována mezi p vláken
- jedno vlákno zvolí pivota a uloží ho do sdílené proměnné
- každé vlákno $i \in 0, \ldots p-1$ rozdělí své prvky na
  - menší než pivot $\rightarrow S_i$
  - větší než pivot $\rightarrow L_i$
- následně je potřeba všechny prvky přeskládat do pomocného pole, aby v něm nejrpve byly prvky menší a potom větší než pivot
- to se provádí pomocí dvou sdílených proměných `s` a `l`
- nejrpve se nastaví `s=0` a `l=n-1`, kde `n` je prvků posloupnosti

Quicksort pro architektury se sdílenou pamětí

- jakmile některé vlákno provede rozělení prvků na menší a větší než pivot provede
  - `smaller = atomicAdd( s,`$|S_i|$`)`
  - `larger = atomicSubtract( l,`$|L_i|$`)`
- zde funkce `atomicAdd` zvětší příslušnou proměnnou a vrátí její původní hodnotu, podobně `atomicSubtract`
- každé vlákno pak překopíruje menší prvky do pomocného pole od pozice `smaller` a větší od pozice `larger` v sestupném pořadí
- následně se obě pole prohodí a vlákna si rozdělí prvky menší a větší úměrně jejich počtu
- celý postup se opakuje

Quicksort pro architektury s distribuovanou pamětí

- vybereme si pivota na jednom procesu a pomocí broadcast ho rozešleme všem ostatním procesům
- každé proces $i \in 0, \ldots p-1$ rozdělí své prvky na
  - menší než pivot $\rightarrow S_i$
  - větší než pivot $\rightarrow L_i$
- následně se provede prefix-sum

$$s_i = \sum_{j=0}^{i-1} |S_i|$$

$$l_i = \sum_{j=0}^{i-1} |L_i|$$

- proces i pak posílá své menší prvky do pomocného pole na pozici s_i a větší prvky na pozici l_i
- toto pole je také rozdistribuované

Analýza quicksortu

- $T_S(n) = \theta(n \log n)$
- $T_P(n,p) = \theta\left(\frac{n}{p}\log p + \frac{n}{p}\log\frac{n}{p}\right) = \theta\left(\frac{n}{p}\log n\right)$
- $S(n,p) = p$
- $E(n,p) = 1$
- $C(n,p) = \theta(n \log n) = \theta(T_S(n))$

Hyperquicksort

- zbývá nám dořešit problém s volbou pivota
- paralelní quicksort v této podobě není příliš efektivní
- modifikace zvaná Hyperquicksort postupuje takto
  - před volbou pivota si každý proces sekvenčním quicksortem uspořádá svou posloupnost
  - pak může snadno určit "medián"
  - ze všech "mediánů"se určí "celkový medián"
  - dále probíhá výpočet stejně
  - dělení na vyšší a nižší podposloupnost je nyní pro každý proces rychlejší

Radix sort

- používá se ke třídění celočíselných klíčů
- využívá toho, že čísla můžeme třídit jakoby lexikograficky podle jejich zápisu v soustavě o určitém základu
- jsou dvě možnosti jak třídit
  - nejprve podle nejvíce významné číslice = MSD (most significant digit)
  - nejprve podle nejméně významné číslice = LSD (least significant digit)

Radix sort

- buď R základ se kterým třídíme
- algoritmus používá pomocné pole o velikosti $R + 1$, do kterého napočítává výskyty jednotlivých cifer na dané pozici zápisu
- toto pole se naplní v prvním průchodu
- potom se udělá prefix sum
- tím získáme pozice "přihrádek", do kterých se pak (v pomocném poli) vloží všechny prvky se stejnou cifrou na dané pozici
- v případě MSD se pak rekurzivně třídí jednotlivé "přihrádky"
- u LSD se opět třídí celé pole, ale při zařazování prvků do nových "přihrádek"se musí postupovat shora dolů

Radix sort

MSD Radix sort (R. Sedgewick, Algoritmy v C)

- `l` a `r` udávají meze, mezi kterými třídíme
- `w` udává pozici cifry, podle které třídíme

```
1  void radixMSD( Item& a[], int l, int r, int w )
2  {
3     int i, j, count[ R + 1 ];
4     if( w > bytesword ) return;
5     if( r-l <= M ) return otherSort(a, l, r);
6     for(j=0;j<R;j++) count[j] = 0;
7     for(i=l;i<=r;i++) count[ digit( a[i], w ) + 1 ]++;
8     for(j=1;j<R;j++) count[j] += count[j-1];
9     for(i=l;i<=r;i++) aux[count[digit(a[i],w)]++] = a[i];
10    switch( a, aux );
11    radixMSD( a, l, count[0] + l-1, w+1 );
12    for(j=0;j<R-1;j++)
13       radixMSD( a, count[j] + l-1, count[ j+1 ] + l-1, w+1 );
14 }
```

Radix sort

LSD Radix sort (R. Sedgewick, Algoritmy v C)

- `l` a `r` udávají meze, mezi kterými třídíme
- `w` udává pozici cifry, podle které třídíme

```
1  void radixLSD( Item& a[], int n, int w )
2  {
3     int i, j, count[ R + 1 ];
4     for(w=bytesword-1;w>=0;w--)
5     {
6        for(j=0;j<R;j++) count[j]=0;
7        for(i=0;i<n;i++) count[ digit( a[i], w ) + 1 ]++;
8        for(j=1;j<R;j++) count[j] += count[j-1];
9        for(i=0;i<n;i++) aux[count[digit(a[i],w)]++] = a[i];
10       swap(a, aux);
11    }
12 }
```

Radix sort

- $T_S(n) = R \cdot n$
- pro $R = 2$ se MSD radix sort podobá quicksortu

Budeme se zabývat paralelizací LSD varianty.

Radix sort pro architektury se sdílenou pamětí

```
1
2  void radixLSD( Item& a[], int n, int w )
3  {
4     int threadsNum = omp_get_num_threads();
5     int pid = omp_get_thread_num();
6     int i, j, count[ R+1 ][threadsNum];
7     #pragma omp parallel shared(a,aux,count)
8     for(w=bytesword-1;w>=0;w--)
9     {
10       int localAccum[ R+1 ];
11       for(j=0;j<R;j++) count[j][pid]=localAccum[j]=0;
12
13       #pragma omp for private(i), schedule (static)
14       for(i=0;i<n;i++) count[ digit( a[i], w ) + 1 ][pid]++;
15
16       partialSum(count, localAccum);
17
18       #pragma omp for private(i), schedule (static)
19       for(i=0;i<n;i++) aux[localAccum[digit(a[i],w)]++] = a[i];
20
21       #pragma omp single
22       swap(a, aux);
23    }
24 }
```

Radix sort pro architektury se sdílenou pamětí

- program používá pole `count`, kam se napočítávají histogramy podle cifer pro jednotlivé procesory
- pole `localAccum` pak obsahuje meze "přihrádek"pro jednotlivé cifry opět pro každý procesor zvlášť – napočítá se podle vztahu

$$localAcc[i] = \sum_{ip=0}^{p-1} \sum_{j=0}^{i-1} count[j][ip] + \sum_{ip=0}^{pid} count[i][ip], 0 \leq i < R$$

- pak se prvky správně přerovnají do pole `aux`

Radix sort pro architektury se sdílenou pamětí

Co je špatně na tomto algoritmu?

Radix sort pro architektury se sdílenou pamětí

Co je špatně na tomto algoritmu?

- uspořádání pole `count`
  - píšeme-li v C `count[j][pid]`, pak dvě po sobě jdoucí vlákna vlákna přistupují k po sobě jdoucím prvkům, což způsobuje false sharing
- řešením je změna na `count[pid][j]` nebo ještě lépe, toto pole rozdělit na několik samostatných, pro každé vlákno zvlášť

Radix sort pro architektury se sdílenou pamětí

```
1
2  void radixLSD( Item& a[], int n, int w )
3  {
4     int threadsNum = omp_get_num_threads();
5     int pid = omp_get_thread_num();
6     int i, j, count[threadsNum][R+1];              // !!!
7     #pragma omp parallel shared(a,aux,count)
8     for(w=bytesword-1;w>=0;w--)
9     {
10       int localAccum[ R+1 ];
11       for(j=0;j<R;j++) localAccum[j]=0;                  // !!!
12
13       #pragma omp for private(i) schedule (static)
14       for(i=0;i<n;i++) count[pid][ digit( a[i], w ) + 1 ]++; // !!!
15
16       partialSum(count, localAccum);
17
18       #pragma omp for private(i) schedule (static)
19       for(i=0;i<n;i++) aux[localAccum[digit(a[i],w)]++] = a[i];
20
21       #pragma omp single
22       swap(a, aux);
23    }
24 }
```

Radix sort pro architektury se sdílenou pamětí

Výsledky naměřené na SGI Origin 2000:

- třídění 16 milionů prvků
- přístup do RAM trvá 75 cyklů CPU

	L_2 misses	Invalidations	Time
Alg. 1	9 828 950	9 725 467	51.29
Alg. 2	1 534 000	251 744	5.76

Radix sort pro architektury se sdílenou pamětí

Co je špatně na algoritmu 2?

Radix sort pro architektury se sdílenou pamětí

Co je špatně na algoritmu 2?

- máme špatnou datovou lokalitu
- v každé iteraci (při třídění s více významnou cifrou) se prvky přehazují napříč celým polem
- to znamená, že v další iteraci je bude přejímat jiné vlákno a navíc tím klesá efektivita využití keše

Radix sort pro architektury se sdílenou pamětí

Co je špatně na algoritmu 2?

- máme špatnou datovou lokalitu
- v každé iteraci (při třídění s více významnou cifrou) se prvky přehazují napříč celým polem
- to znamená, že v další iteraci je bude přejímat jiné vlákno a navíc tím klesá efektivita využití keše
- je potřeba zvýšit datovou lokalitu a k tomu se lépe hodí MSD varianta
- ta je bohužel náročnější na paralelizaci

Radix sort pro architektury se sdílenou pamětí

- řešením je provést první krok MSD ...
  - ten provedeme stejně jako u paralelní LSD, tedy paralelně
- dále se všechny přihrádky rozdělí mezi procesory
- protože žádný prvek se už nebude přesouvat do jiné přihrádky, nebude docházet k přesunům mezi vlákny a procesy
- navíc s tím, jak třídíme stále menší a menší přihrádky dostáváme lepší využití keše
- tím dostáváme Algoritmus 3

Radix sort pro architektury se sdílenou pamětí

Dostáváme následující výsledek:

	L_2 misses	Invalidations	Time
Alg. 1	9 828 950	9 725 467	51.29
Alg. 2	1 534 000	251 744	5.76
Alg. 3	833 559	311 237	2.55

Radix sort pro architektury s distribuovanou pamětí

```
1  void radixLSD( Item& a[], int n, int w )
2  {
3     int i, j, count[p][R+1];
4     for(w=bytesword-1;w>=0;w--)
5     {
6        for(j=0;j<R;j++) count[pid][j]=0;
7        for(i=0;i<n;i++) count[pid][ digit( a[i], w ) + 1 ]++;
8        for(j=1;j<R;j++) count[pid][j] += count[pid][j-1];
9        for(i=0;i<n;i++) aux[count[pid][digit(a[i],w)]++] = a[i];
10
11       allgather(count[pid][0],R,count);
12
13       bucketDistribuiton(count);
14       dataCommunication(a,aux,count)
15       swap(a, aux);
16    }
17 }
```

Radix sort pro architektury s distribuovanou pamětí

- algoritmus nejprve provede lokálně klasicky jednu iteraci radix sortu
- v poli `count[pid]` má uložené rozmezí přihrádek
- funkce `allgather` způsobí, že všechny uzly budou mít kompletní pole `count`
- následně se pomocí něj napočítají celkové velikosti přihrádek včetně toho že některé přihrádky se mohou rozdělit na dva uzly pro lepší vyvážení zátěže – `bucketDistribution`
- nakonec proběhne výměna dat – `dataCommunication`

Radix sort pro architektury s distribuovanou pamětí

Implementace datové komunikace:

- zde se dobře hodí funkce `alltoallv`

Problém tohoto přístupu je opět v tom, že prakticky všechny prvky tříděného pole se v každém kroku přesouvají z jednoho uzlu na druhý.

Radix sort pro architektury s distribuovanou pamětí

- řešením je opět využití MSD varianty
- po první iteraci se přihrádky rozdělí mezi jednotlivé uzly
- pak už mezi nimi neprobíhá žádná komunikace
- ve výsledku jsme schopni dostat stejně efektivní algoritmus jako na pro architekturu se sdílenou pamětí

Třídící sítě

- provádějí velký počet porovnání najednou
  - většinou tolik, kolik je prvků tříděné posloupnosti
- k tomu používají tzv. **komparátory**

Definition

Komparátor je zobrazení uspořádané dvojice (x, y) na jinou uspořádanou dvojici (x', y'). Pro **rostoucí komparátor** platí

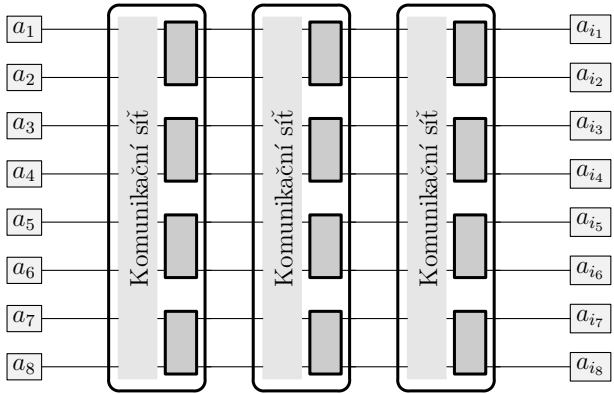
$$x' = \min\{x, y\} \quad y' = \max\{x, y\},$$

a budeme ho označovat pomocí $\oplus$. Pro **klesající komparátor** platí

$$x' = \max\{x, y\} \quad y' = \min\{x, y\},$$

a budeme ho označovat pomocí $\ominus$.

Třídící sítě



Bitonic sort

Definition

Levé cyklické posunutí je operace, která transformuje posloupnost

$$\{a_1, \ldots, a_n\}$$

na posloupnost

$$\{a_2, \ldots, a_n, a_1\}.$$

Pravé cyklické posunutí pak tranformuje posloupnost

$$\{a_1, \ldots, a_n\}$$

na posloupnost

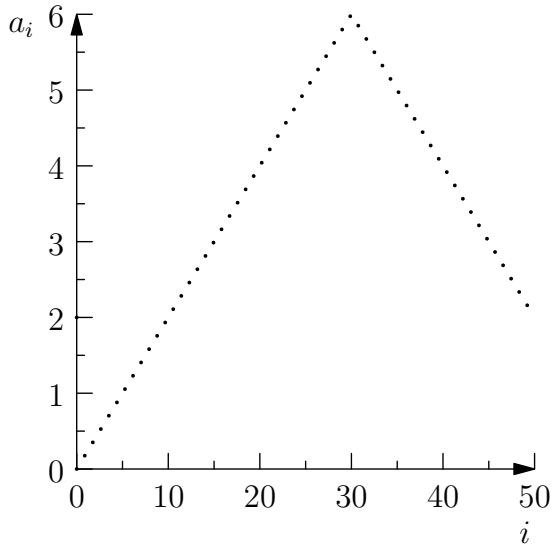
$$\{a_n, a_1, \ldots, a_{n-1}\}.$$

Bitonic sort

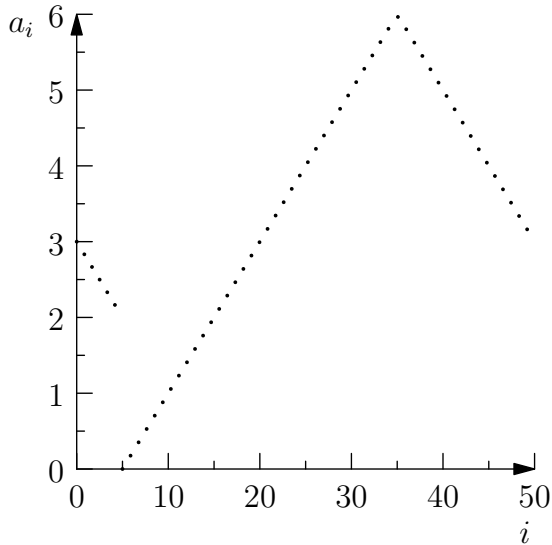
Definition

Bitonická posloupnost je taková, která se skládá z jedné rostoucí podposloupnosti a jedné klesající nebo tohoto lze dosáhnout aplikací libovolného počtu levých nebo pravých cyklických posunutí..

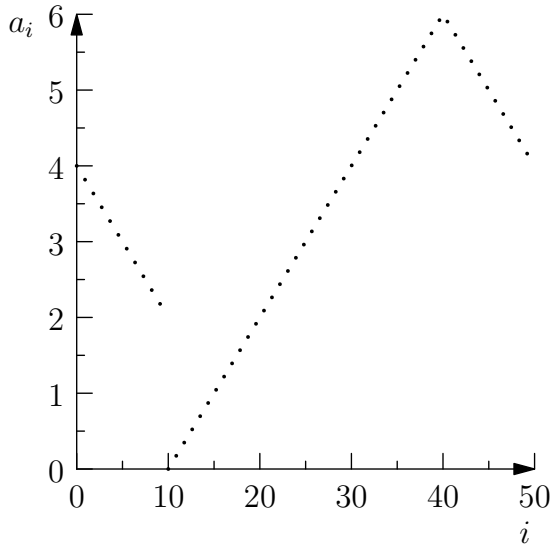
Bitonic sort



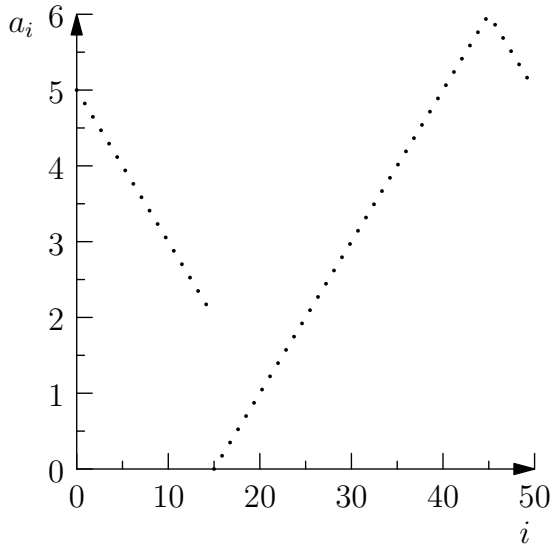
Bitonic sort



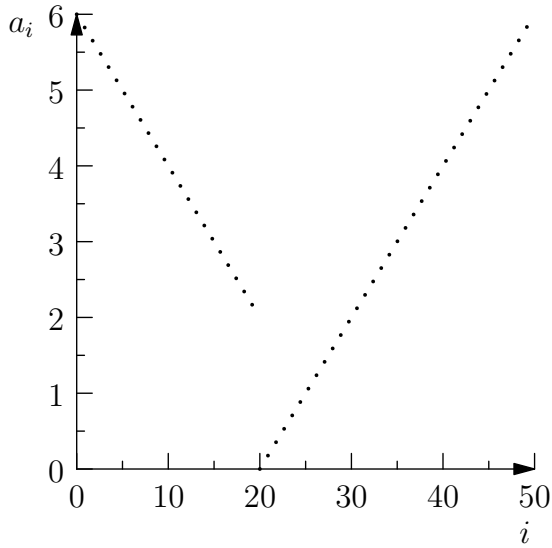
Bitonic sort



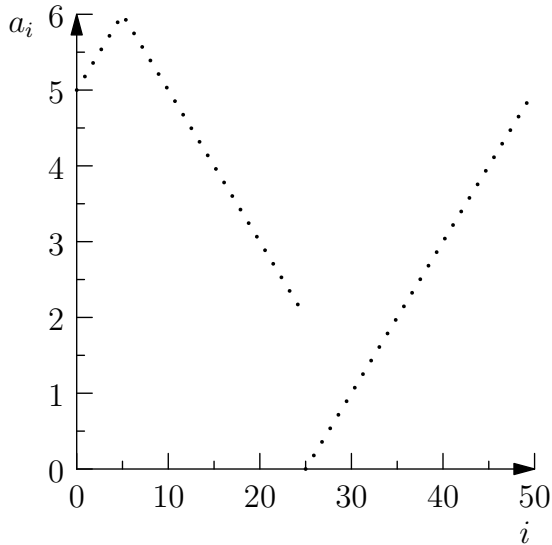
Bitonic sort



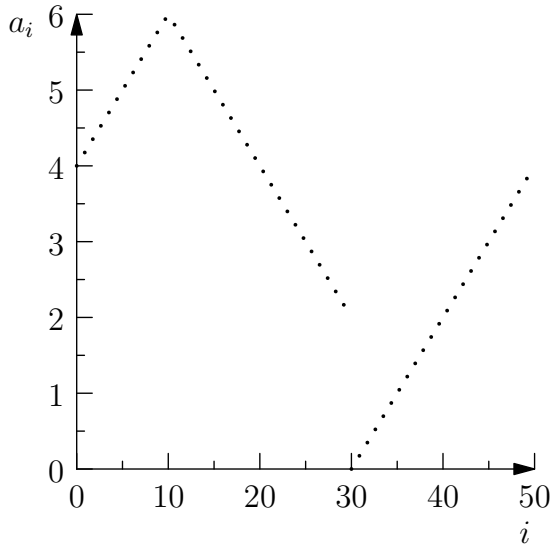
Bitonic sort



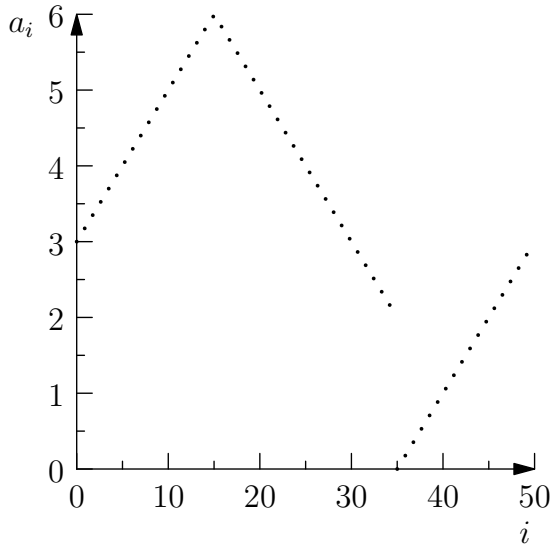
Bitonic sort



Bitonic sort



Bitonic sort



Bitonic sort

Definition

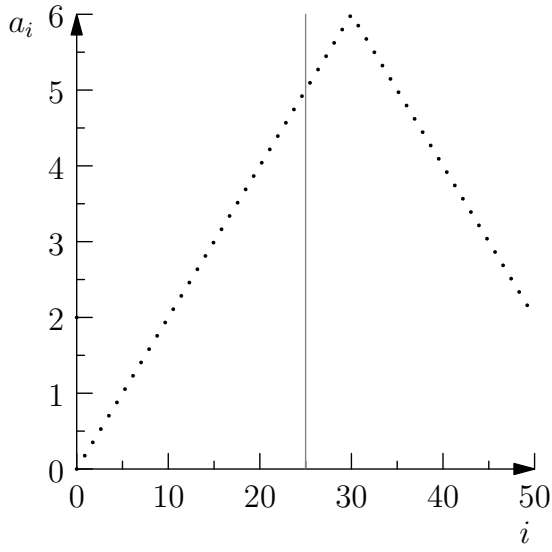
Bitonické rozdělení (*bitonic split*) je operace, která z posloupnosti

$$\{a_1, \ldots, a_{2n}\}$$

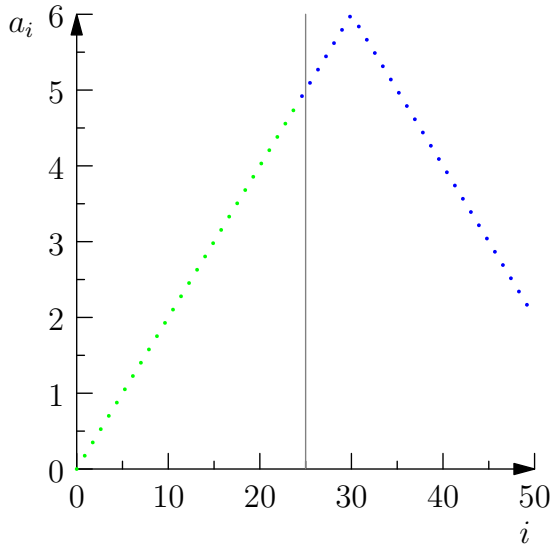
vytvoří své poslouposti

$$\begin{aligned} s_{min} &= \{\min\{a_0, a_n\}, \min\{a_1, a_{n+1}\}, \ldots, \min\{a_{n-1}, a_{2n-1}\}\}, \\ s_{max} &= \{\max\{a_0, a_n\}, \max\{a_1, a_{n+1}\}, \ldots, \max\{a_{n-1}, a_{2n-1}\}\}. \end{aligned}$$

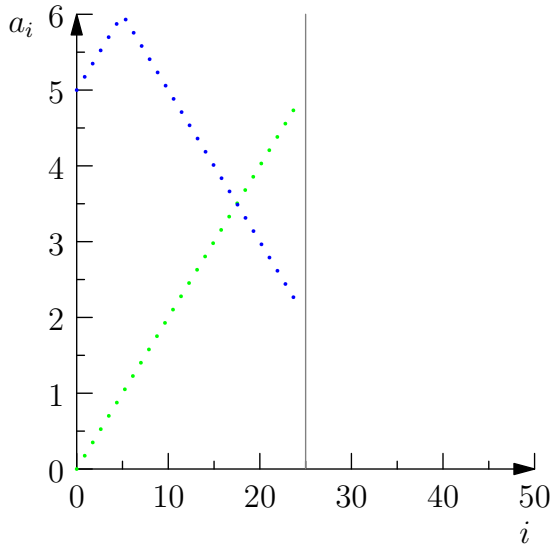
Bitonic sort



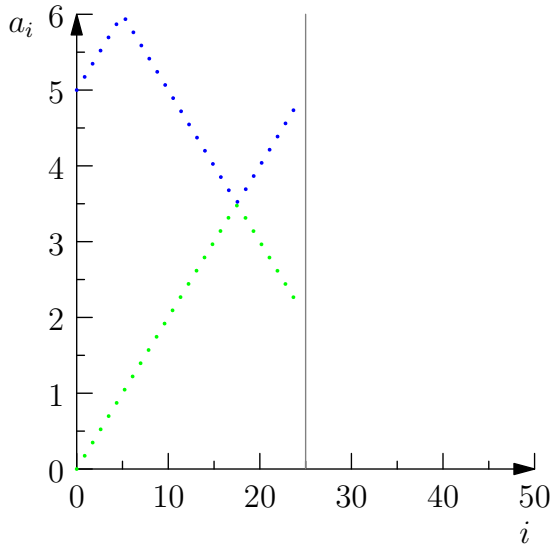
Bitonic sort



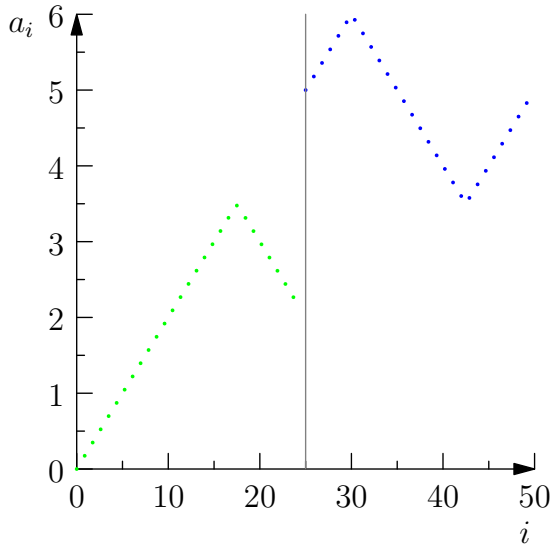
Bitonic sort



Bitonic sort



Bitonic sort



Bitonic sort

Lemma

Pokud provedeme levé nebo pravé cyklické posunutí s posloupností $\{a_1, \ldots, a_{2n}\}$ *projeví se stejné cyklické posunutí i na posloupnostech* s_{min} *a* s_{max}*.*

Lemma

Otočení posloupnosti $\{a_1, \ldots, a_{2n}\}$ *na posloupnost* $\{a_{2n}, \ldots, a_1\}$ *zachovává bitonicitu, tj.* $\{a_{2n}, \ldots, a_1\}$ *je bitonická, pokud* $\{a_1, \ldots, a_{2n}\}$ *je bitonická. Posloupnosti* s_{min} *a* s_{max} *se otočí stejným způsobem.*

Theorem

Posloupnosti s_1 *a* s_2 *z bitonického dělení jsou obě bitonické a platí*

$$\max\{s_{min}\} \leq \min\{s_{max}\}.$$

Bitonic sort

Důkaz:

Díky předchozím lematům stačí dokázat větu pouze pro situaci, kdy je:

$$a_1 \leq a_2 \leq \ldots \leq a_j \geq a_{j+1} \geq \ldots \geq a_{2n}.$$

Mohou nastat dvě situace:

Situace 1: $a_n \leq a_{2n}$, potom $a_i \leq a_{i+n}$ pro $1 \leq i \leq n$ a proto
$s_{min,i} = \min\{a_1, a_{n+1} = a_i\}$ a $s_{max,i} = \max\{a_1, a_{n+1}\} = a_{n+i}$ a proto

$$\max\{s_{min}\} = a_n \leq \min\{s_{max}\}.$$

Bitonic sort

Sitace 2:
$a_n \geq a_{2n}$, pak existuje k takové, že $j \leq k \leq 2n$ a $a_{k-n} \leq a_k$ a $a_{k-n+1} > a_{k+1}$.
Potom je

$$\left.\begin{array}{c} s_{min,i} = a_i \\ s_{max,i} = a_{i+n} \end{array}\right\} \text{ pro } 1 \leq i \leq k-n \text{ a } \left.\begin{array}{c} s_{min,i} = a_{i+n} \\ s_{max,i} = a_i \end{array}\right\} \text{ pro } k-n+1 < i \leq n.$$

- s_{min} je rostoucí pro $i = 1, \ldots, k-n$ a klesající pro $i = k-n, \ldots, n$, takže je bitonická.
- s_{man} je klesající pro $i = 1, \ldots, k-n$ a rostoucí pro $i = k-n, \ldots, n$, takže je bitonická.
- $\max\{s_{min}\} = \max\{a_{k-n}, a_{k+1}\}$ a $\min\{s_{max}\} = \min\{a_{k-n+1}, a_k\}$.
- Je-li $\max\{s_{min}\} = a_{k-n}$ pak je $a_{k-1} \leq a_{k-n+1}$, protože ze je a_n rostoucí a $a_{k-n} \leq a_k$ z definice k.
- Je-li $\max\{s_{min}\} = a_{k+1}$, potom $a_{k+1} < a_{k-n+1}$ podle definice k a $a_{k+1} \leq a_k$, protože a_n je zde klesající. □

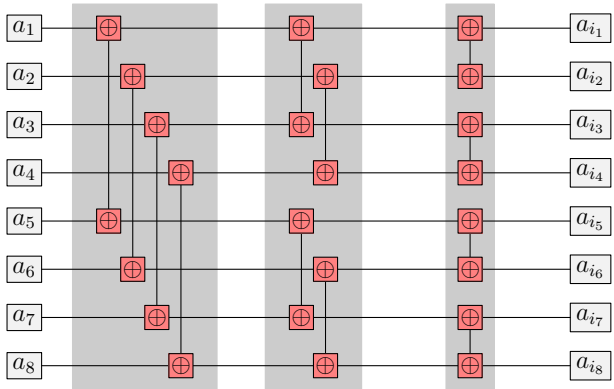
Bitonic sort

Podle předchozí věty:

- jsme získali dvě bitonické posloupnosti
- všechny prvky posloupnosti s_{min} jsou menší, než všechny prvky posloupnosti s_{max}
- Obě posloupnosti mohu dále třídit nezávisle, tj. paralelně.
- Na obě posloupnosti použiji rekurzivně stejný postup.
- Tak nakonec dojdu k posloupnostem délky 1.

Algoritmus nyní zakreslíme pomocí komparátorů.

Bitonic sort

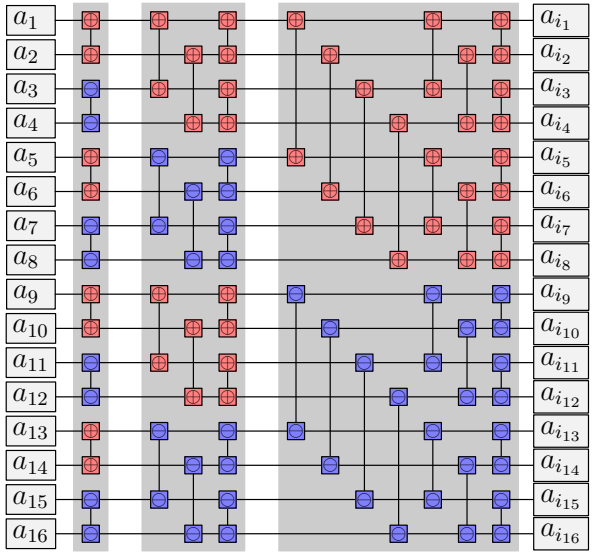


Bitonic sort

Jak ale postupovat, pokud nemáme bitonickou posloupnost?

- Vycházíme z faktu, že každá posloupnost délky dva je bitonická (je roustoucí nebo klesající).
- Následující obrázek ukazuje, jak slučovat více bitonických posloupností do jedné jediné.

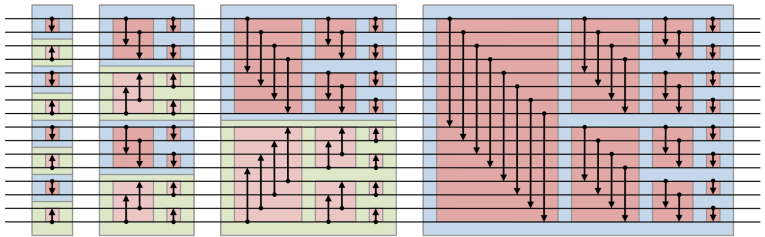
Bitonic sort



Bitonic sort

- postup je založen na skládání dvou bitonických posloupností do jedné o dvojnásobné délce
- začínáme s posloupnostmi o délce 1
- z těch snadno získáme bitonické posloupnosti o délce 2
- pro ukázku si ukážeme, jak získat bitonickou posloupnost o délce 4

Bitonic sort



Obrázek: Zdroj: Wikipedia

Enumeration sort

Jde o teoretický algoritmus pro CRCW PRAM.

- mějme n^2 procesorů indexovaných indexy $i, j = 1, \ldots n$
- nechť CRCW PRAM používá při zápisu sčítací protokol

```
procedure enumSort( A, n )
begin
    for each P_1j do C[ j ] := 1;
    for each P_ij do
       if( A[i]<A[j] ) or ( ( A[i] = A[j] ) and ( i < j ) ) then
          C[j] := 1;
    for each P_1j do A[C[j]] := A[ j ];
end
```