

Paralelní třídění

Úvod

Bublínkové třídění

Quicksort

Radix sort

Třídící algoritmy

Vstup: Posloupnost prvků a_1, \dots, a_n s definovaným uspořádáním \leq .

Výstup: Setříděná posloupnost, pro kterou platí $a_{i_1} \leq a_{i_2} \dots a_{i_n}$.

- ▶ 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

Bublňkové řídění

Sekvenční algoritmus:

- ▶ porovnááme a uspořádváme postupně prvky:

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

- ▶ 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.

$$(a_1, a_2), (a_3, a_4), \dots (a_{n-1}, a_n)$$
$$(a_2, a_3), (a_4, a_5), \dots (a_{n-2}, a_{n-1})$$

- ▶ 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

- ▶ $T_P(n, p) = \frac{n}{2} \frac{n-1}{p} = \theta\left(\frac{n^2}{2p}\right)$
- ▶ $S(n, p) = \frac{n^2}{2} \frac{2p}{n^2} = p$
- ▶ $E(n, p) = 1$
- ▶ $C(n, p) = \frac{n^2}{2} = \theta(T_S(n^2))$
- ▶ 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
 - ▶ $a_1, a_2, \dots, a_{\frac{n}{2}-1}$ a $a_{\frac{n}{2}}, \dots, a_n$
 - ▶ 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 rámci 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

$$a_1, \dots, a_m$$

na dvě

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

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 rozdistributedována mezi p vláken
- ▶ jedno vlákno zvolí pivota a uloží ho do sdílené proměnné
- ▶ každé vlákno $i \in 0, \dots, p - 1$ 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 nejprve byly prvky menší a potom větší než pivot
- ▶ to se provádí pomocí dvou sdílených proměnných s a l
- ▶ nejprve se nastaví $s=0$ a $l=n-1$, kde n je prvků posloupnosti

Quicksort pro architektury se sdílenou pamětí

- ▶ jakmile některé vlákno provede rozdělení prvků na menší a větší než pivot provede
 - ▶ `smaller = atomicAdd(s, |Si|)`
 - ▶ `larger = atomicSubtract(l, |Li|)`
- ▶ 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 $i \in 0, \dots, p - 1$ rozdělí své prvky na
 - ▶ menší než pivot $\rightarrow S_i$
 - ▶ větší než pivot $\rightarrow L_i$
- ▶ následně se provede prefix-sum

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

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

- ▶ 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 roz distribuované

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

Quicksort na GPU

Cederman, Tsigas, *A Practical Quicksort Algorithm for Graphics Processors*, 2008.

Navrhují následující postup:

- ▶ v první fázi jsou zpracovávány velké podposloupnosti
 - ▶ na každou podposloupnost se mapuje několik CUDA bloků vláken
 - ▶ je proto potřeba dát pozor na správnou synchronizaci
 - ▶ protože potřebuje synchronizovat CUDA bloky, je nutné ukončit jeden kernel, a pak spustit další
- ▶ v druhé fázi už je jednotlivých nezávislých podposloupností dost na to, aby se každému CUDA bloku přiřadila jedna
 - ▶ tím odpadá problém se synchronizací mezi CUDA bloky
 - ▶ k dalšímu dělení v rámci jednoho bloku se používá explicitní zásobník
 - ▶ na třídění malých podposloupností se používá jiný třídící algoritmus
- ▶ algoritmus nepoužívá in-place třídění, protože tím by se špatně dosahovalo sloučených přístupů do globální paměti

Quicksort na GPU

První fáze:

- ▶ máme danou posloupnost a pivota
- ▶ posloupnost chceme rozdělit na prvky menší, než je pivot, a prvky větší
- ▶ posloupnost rozdělíme na m bloků a pro každý blok pustíme jeden CUDA blok

Quicksort na GPU

- ▶ každé vlákno CUDA bloku načte jeden prvek a porovná, zda je větší nebo menší, než pivot
- ▶ ve sdílené paměti jsou alokována dvě pomocná pole `smaller` a `larger` o velikosti rovné počtu vláken v CUDA bloku
- ▶ každé vlákno na svou pozici přičte jedna v poli `smaller` má-li prvek menší než pivot, nebo přičte jedna do pole `larger`, má-li prvek větší než pivot
- ▶ po projití všech dat přiřazených CUDA bloku se provede prefix sum na obě pole
- ▶ získáme tak offsety prvků příslušejících jednotlivým CUDA vláknům ve výsledné posloupnosti - ale jen v rámci jednotlivých CUDA bloků

Operace FAA

- ▶ nyní potřebujeme offsety ve výsledné posloupnosti jednotlivých CUDA bloků
- ▶ k tomu se použije atomická funkce FAA (fetch-and-add)

```
int atomicAdd( int* address, int val);
```

- ▶ tato funkce atomicky přičte k hodnotě `old` na adrese `adrese` hodnotu `val`, výsledek zapíše na stejnou adresu a vrátí původní hodnotu `old`
- ▶ my tuto instrukci využijeme tak, že si v globální paměti uděláme proměnnou udávající celkový počet prvků menších než `pivot` a proměnnou pro celkový počet prvků větší, než `pivot`
- ▶ každý CUDA blok, který má napočítané větší a menší prvky, přičte atomicky své počty větších a menších prvků a zároveň použije původní hodnoty jako své offsety

Quicksort na GPU

- ▶ nyní již může každý blok projít znovu vstupní data a roztrídít je na požadované pozice ve výstupním pomocném poli
 - ▶ prvky menší než pivot vkládáme podle offsetu zleva
 - ▶ prvky větší než pivot vkládáme podle offsetu zprava
- ▶ nakonec jeden nebo každý blok zapíše pivota na správné místo

Quicksort na GPU

- ▶ pokud je některá z výsledných podposloupností větší, než `minLength`, vybereme v ní `pivota` a určíme ji k dalšímu dělení posloupnost
- ▶ pokud již máme více podposloupností než `maxSequences` přejdeme do fáze 2
- ▶ druhá fáze nemusí vzájemně synchronizovat různé CUDA bloky

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 "příhrádek", do kterých se pak (v pomocném poli) vloží všechny prvky se stejnou cifrou na dané pozici
- ▶ v případě MSD se pak rekurzivně třídí jednotlivé "příhrádky"
- ▶ u LSD se opět třídí celé pole, ale při zařazování prvků do nových "příhrádek" se musí postupovat shora dolů

Radix sort

MSD Radix sort (R. Sedgwick, 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=1;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. Sedgwick, 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 "příhrádek" 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 síť

- ▶ 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

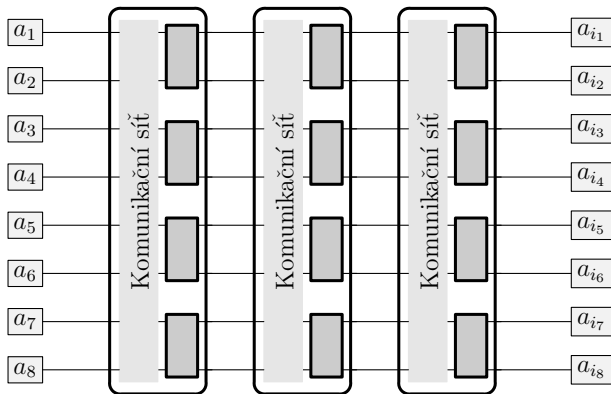
$$x' = \min \{x, y\} \quad y' = \max \{x, y\},$$

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 síť



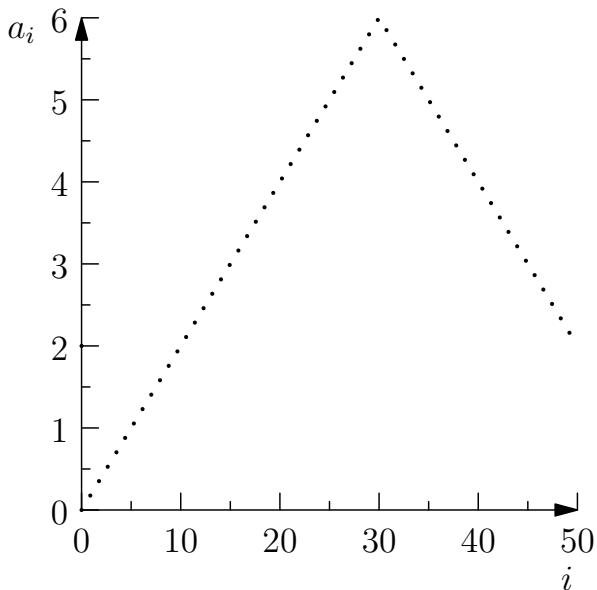
Bitonic sort

- ▶ bitonic sort je založený na třídění **bitonických posloupností**

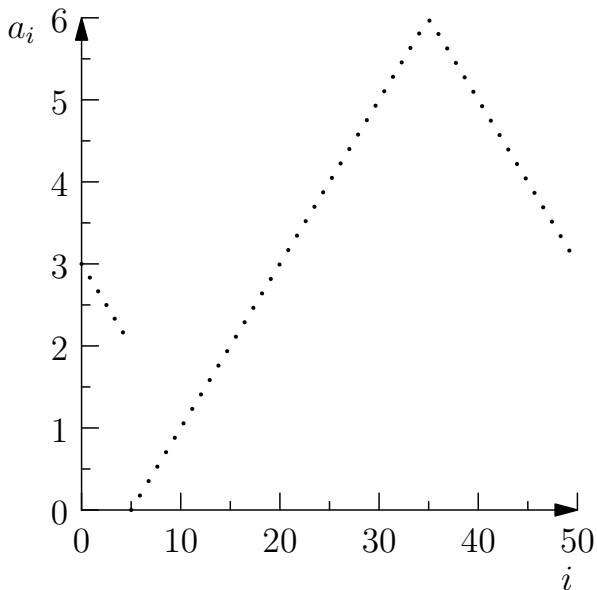
Definition

Bitonická posloupnost je taková, která se skládá z jedné rostoucí podposloupnosti a jedné klesající nebo tohoto lze dosáhnout rotací.

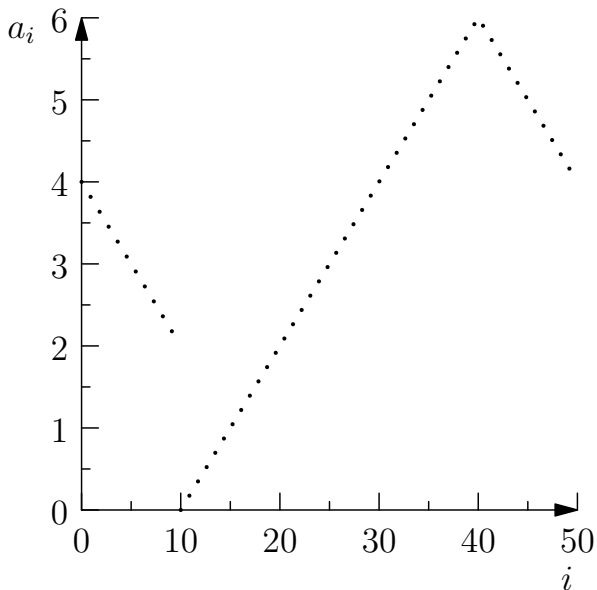
Bitonic sort



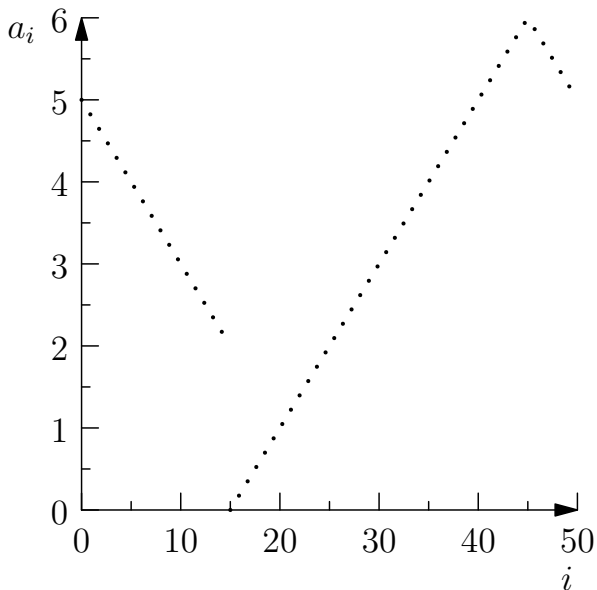
Bitonic sort



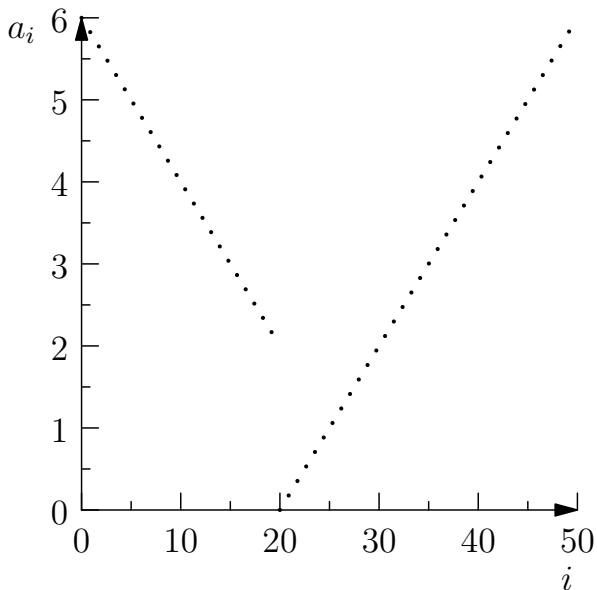
Bitonic sort



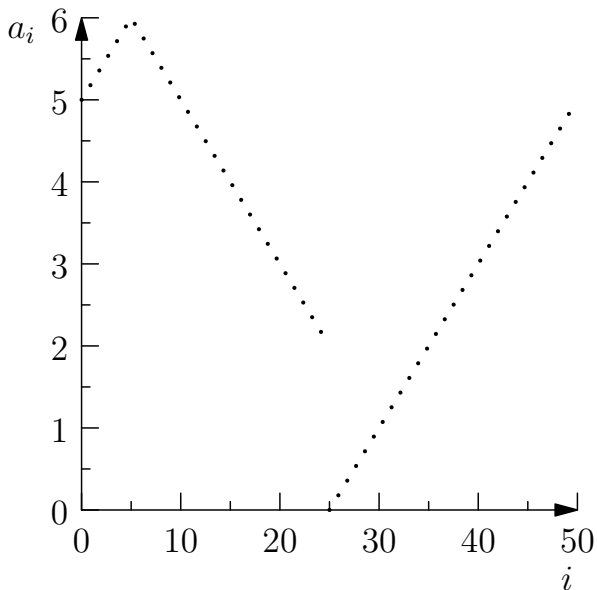
Bitonic sort



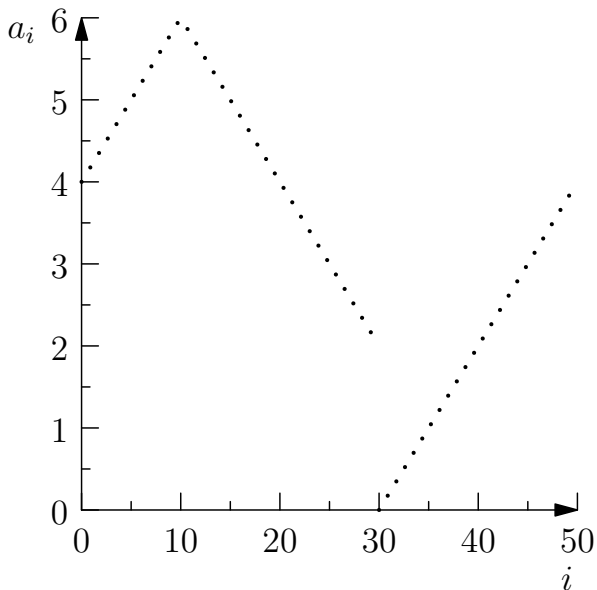
Bitonic sort



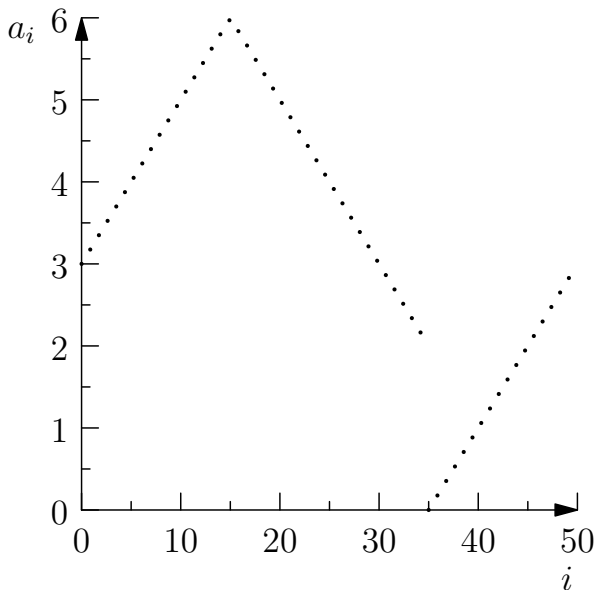
Bitonic sort



Bitonic sort



Bitonic sort



Bitonic sort

Definujme nyní tyto posloupnosti:

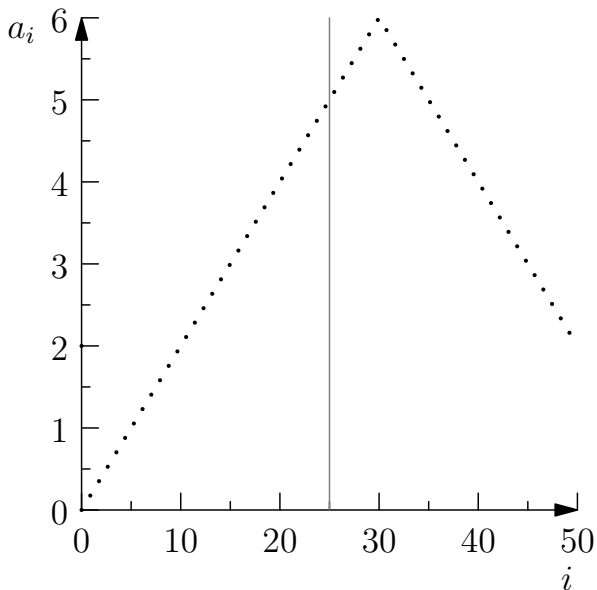


$$s_1 = \left\{ \min \left\{ a_0, a_{\frac{n}{2}} \right\}, \min \left\{ a_1, a_{\frac{n}{2}+1} \right\}, \dots, \min \left\{ a_{\frac{n}{2}-1}, a_{n-1} \right\} \right\}$$

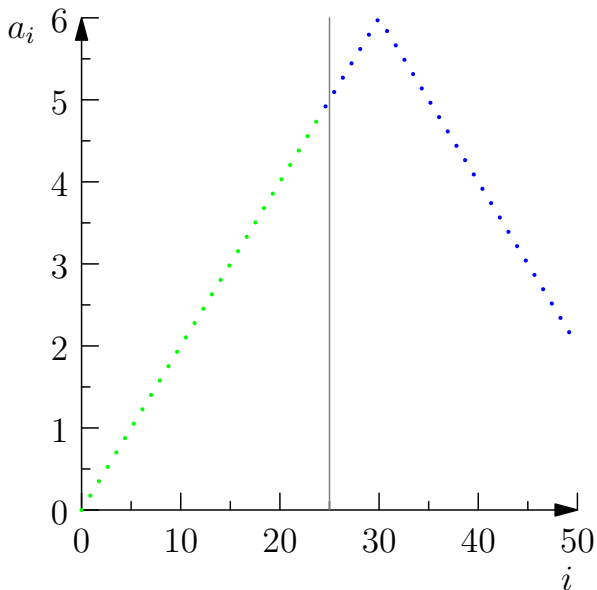


$$s_2 = \left\{ \max \left\{ a_0, a_{\frac{n}{2}} \right\}, \max \left\{ a_1, a_{\frac{n}{2}+1} \right\}, \dots, \max \left\{ a_{\frac{n}{2}-1}, a_{n-1} \right\} \right\}$$

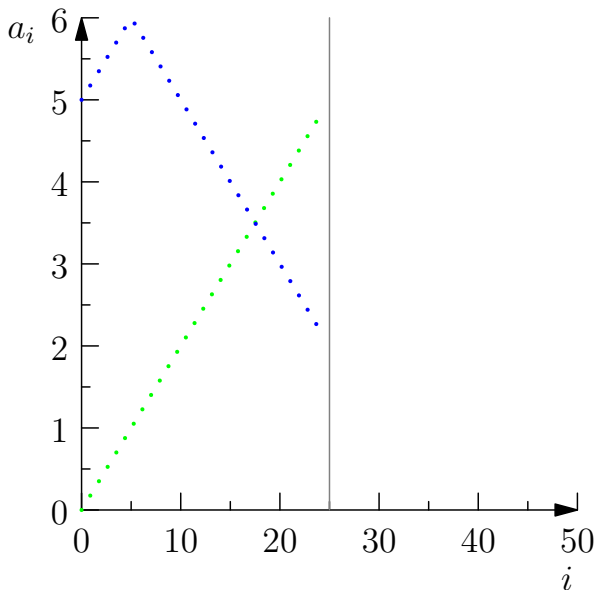
Bitonic sort



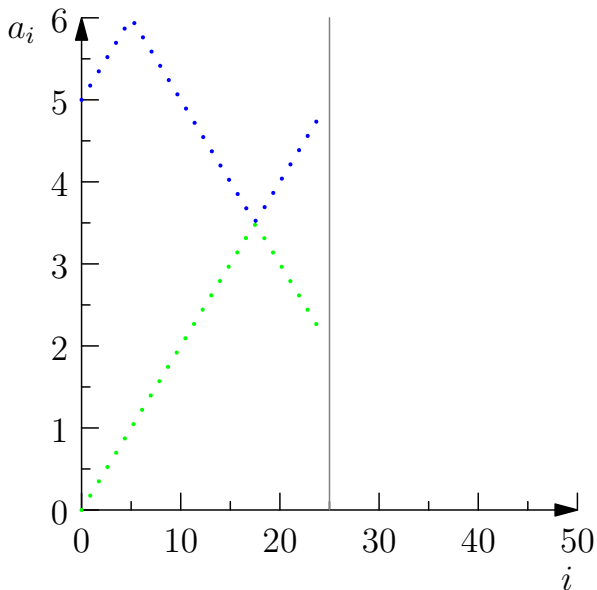
Bitonic sort



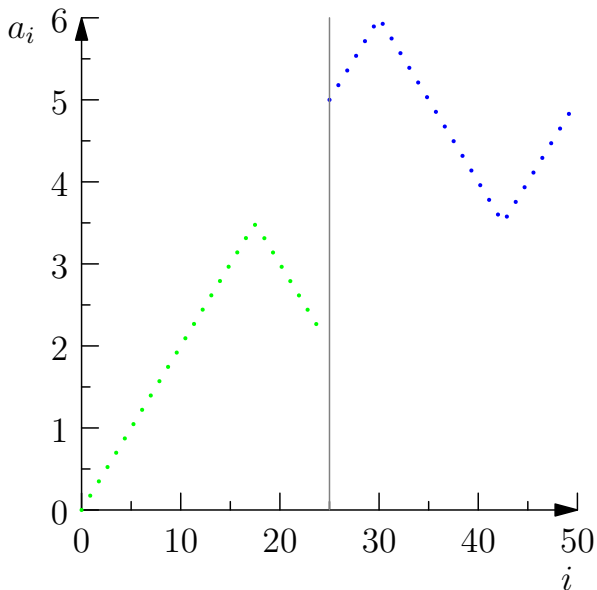
Bitonic sort



Bitonic sort



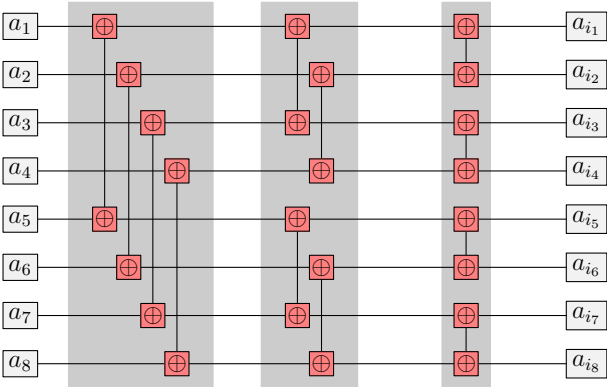
Bitonic sort



Bitonic sort

- ▶ získali jsme tak dvě bitonické posloupnosti
- ▶ navíc platí, že všechny prvky posloupnosti s_1 jsou menší, než prvky posloupnosti s_2
- ▶ mohu tak obě posloupnosti dále třídit nezávisle, tj. paralelně
- ▶ na obě posloupnosti použiji rekurzivně stejný postup
- ▶ tak nakonec dojdou k posloupnostem délky 1
- ▶ algoritmus nyní zakreslíme pomocí komparátorů

Bitonic sort

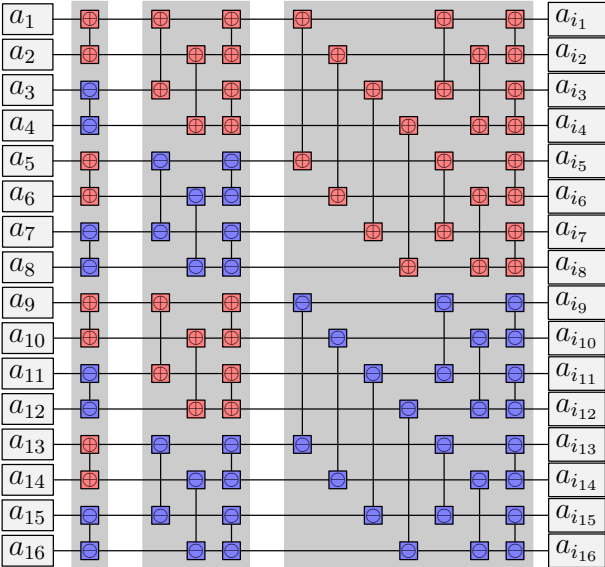


Bitonic sort

Jak ale postupovat, pokud nemáme bitonickou posloupnost?

- ▶ následující obrázek ukazuje, jak jí získat i z obecné posloupnosti

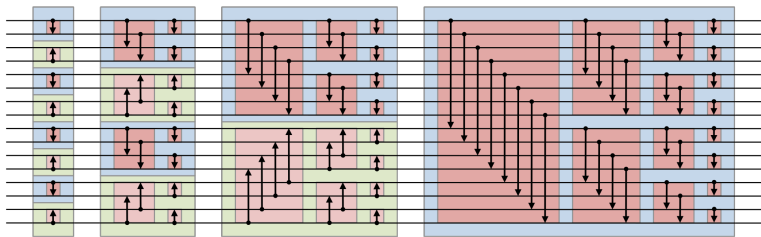
Bitonic sort



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 ukázkou 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 $i, j = 1, \dots, n$
- ▶ necht' CRCW PRAM používá při zápisu sčítací protokol

```
procedure enumSort( A, n )
begin
  for each  $P_{ij}$  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  $P_{ij}$  do A[C[j]] := A[ j ];
end
```