Paralelní třídění

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 je triviální
  - 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 je také triviální
  - 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 je také celkem jednoduchá
  - pole je stále v globální paměti
    - po každé iteraci je nutné synchronizovat všechna vlákna v rámci gridu
  - každé vlákno načte jeden prvek do sdílené paměti
  - sudá vlákna pak provedou porovnání a prohození ve sdílené paměti
  - nakonec každé vlákno zapíše jeden prvek do celého pole
- to zřejmě nebude moc efektivní
- třídění malých posloupností v rámci jednotlivých bloků by ale mohlo být schůdné

Shellovo třídění

- algoritmus funguje tak, že třídí pole s krokem h
  - tj. když vezmeme každý h-tý prvek, získáme setříděnou posloupnost
- tím pádem získáváme h nezávislých podposloupností, které lze třídit současně
- na třídění se používá metoda vkládání
- parametr h zmenšujeme až k 1

Shellovo třídění - paralelně

- stupeň paralelizace je závislý na h
- s tím, jak se h zmenšuje, klesá i efektivita paralelizace
- podposloupnosti se mění s každým novým h, takže algoritmus není vhodný pro architektury s distribuovanou pamětí
  - při každé změně h bych musel dělat scatter a gather
- podposloupnosti jsou vzájemně provázané, takže u architektur se sdílenou pamětí hrozí false sharing, neboť různé procesy budou načítat sousední prvky
- na GPU lze jen těžko dosáhnout sloučených přístupů do paměti, u sdílené paměti by při určitých krocích mohlo docházet ke konfliktům v přístupu do paměťových bank
- Shellovým tříděním se dál zabývat nebudeme

Přihrádkové třídění - bucket sort

- předpokládejme, že třídíme reální čísla z intervalu $[0, 10]$
- celý interval si můžeme rozdělit na 10 přihrádek a velikosti 1
- do i-té přihrádky vložím všechna čísla a_j taková, že

$$i - 1 \leq a_j < i$$

- každou přihrádku pak setřídím zvlášť libovolným třídícím algoritmem
- výsledné setříděné podposloupnosti poskládám za sebe
- aby byl algoritmus efektivní, je potřeba mít všechny přihrádky zaplněné rovnoměrně
- složitost silně závisí na zaplnění přihrádek
  - pokud padnou všechny prvky do jedné, bude složitost dána přesně použitým třídícím algoritmem
  - pokud bych věděl, že do každé přihrádky padne pravě jeden prvek, vkládám prvky vlastně rovnou na své místo a mám složitost $O(n)$

Přihrádkové třídění - paralelně

- paralelizovat lze snadno jak rozdělování do přihrádek ...
- ... tak i vlastní třídění ...
- ... a následné složení do setříděné posloupnosti

Přihrádkové třídění - paralelně

Implementace na architekturách se sdílenou pamětí

- vstupní posloupnost se rozdělí rovnoměrně blokově mezi p procesorů
- každý procesor prochází svou část a rozděluje do přihrádek
  - vkládání do přihrádek musí být ošetřeno pomocí kritické sekce
- pak se přihrádky rozdělí rovnoměrně mezi všechny procesory
  - zde je dobré vzít v úvahu zaplnění jednotlivých přihrádek včetně složitosti použitého třídícího algoritmu
- provede se setřídění přihrádek
- nakonec zbývá poskládání do výsledného pole
- pomocí operace prefix sum použité na počet prvků, které třídil jeden proces lze přesně určit, kam se do výsledného pole mají ukládat prvky z daného procesu
  - buď m_i počet prvků zpravovaných procesem P_i
  - buď $M_i = \sum_{j=0}^{i-1} m_i$
  - pak proces P_i bude vkládat své prvky na pozice $M_i \ldots M_{i+1}$ ve výsledné setříděné posloupnosti

Přihrádkové třídění - paralelně

Implementace na architekturách s distribuovanou pamětí

- každý uzel dostane několik přihrádek
- vstupní posloupnost rozděluje nultý proces do jednotlivých přihrádek tj. rozesílá jednotlivým uzlům
- každý uzel zařazuje zaslaná data do svých přihrádek, případně je může rovnou zatřiďovat
- přihrádky je možné adaptivně přenášet z jednoho uzlu na druhý tak, aby se zachovalo rovnoměrné vytížení uzlů
- nakonec zbývá poskládání do výsledného pole
- pomocí operace prefix sum použité na počet prvků, které třídil jeden proces lze přesně určit, kam se do výsledného pole mají ukládat prvky z daného procesu
  - buď m_i počet prvků zpravovaných procesem P_i
  - buď $M_i = \sum_{j=0}^{i-1} m_i$
  - pak proces P_i bude vkládat své prvky na pozice $M_i \ldots M_{i+1}$ ve výsledné setříděné posloupnosti

Přihrádkové třídění - paralelně

Implementace na GPU (CUDA)

- v CUDA může být přihrádkové třídění použito jako základ pro jiné třídící algoritmy
- každému CUDA bloku se pak přidělí setřídění jedné přihrádky
- tím lze navýšit paralelismus potřebný pro GPU
- předpokládáme, že data jsou uložena na GPU
- CPU navrhne pivoty, pro dělení do přihrádek, tj. meze jednotlivých přihrádek
- kernel na GPU pak spočítá, kolik prvků připadne do jednotlivých přihrádek
- výsledek pošle na CPU, který může velikosti přihrádek ještě upravit

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 paralelně

- paralelizace 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 paralelně

- předpokládáme tedy, že nesetříděná posloupnost je rozdistribuována v pamětech všech procesů
- vybereme si pivota na jednom procesu a pomocí broadcast ho rozešleme všem ostatním procesům
- každý proces si rozdělí svou podposloupnost na část menší než pivot a část větší než pivot
- každý proces z horní poloviny procesů (podle ID) pošle svou nižší podposloupnost svému protějšku v dolní polovině procesů a naopak
  - u architektur se sdílenou pamětí jde o přeuspořádání celé posloupnosti
  - to, kam se má vložit daná podposloupnost lze opět určit pomocí operace prefix sum
- nakonec mají procesy v dolní polovině nižší podposloupnost a procesy v horní polovině vyšší podposloupnost
- obě mohou být zpracovány rekurzivně dál
- po $\log p$ krocích má každý proces nesetříděnou podposloupnost tak, že všechny prvky na procesech s

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

Analýza hyperquicksortu

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

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 dělní 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 o velikosti jednoho warpu (32 vláken)
- v daném CUDA bloku se musí postupně spustit několik warpů
- každé vlákno daného warpu na svou pozici (podle svého ID) v prvním poli přičte jedna má-li prvek menší než pivot, nebo přičte jedna do druhého počet, má-li prvek větší než pivot
- po projití všech dat přiřazených CUDA bloku se provede prefix sum na obě pole
- získáme tak offsety jednotlivých prvků 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 `adress` 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 roztřídit 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

- algoritmus vyžaduje explicitní implementaci zásobníku
- aby se příliš nerozrůstal, zpracovávají se nejdříve menší podposloupnosti
- pak platí, že zásobník nebude větší než $\log_2 n$
- prefix-sum lze počítat sekvenčně nebo paralelně (asi o 20% rychlejší)
  - M. Harris, S. Sengupta, J. D. Owens, Chapter 39. Parallel Prefix Sum (Scan) with CUDA, GPU Gems 3

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

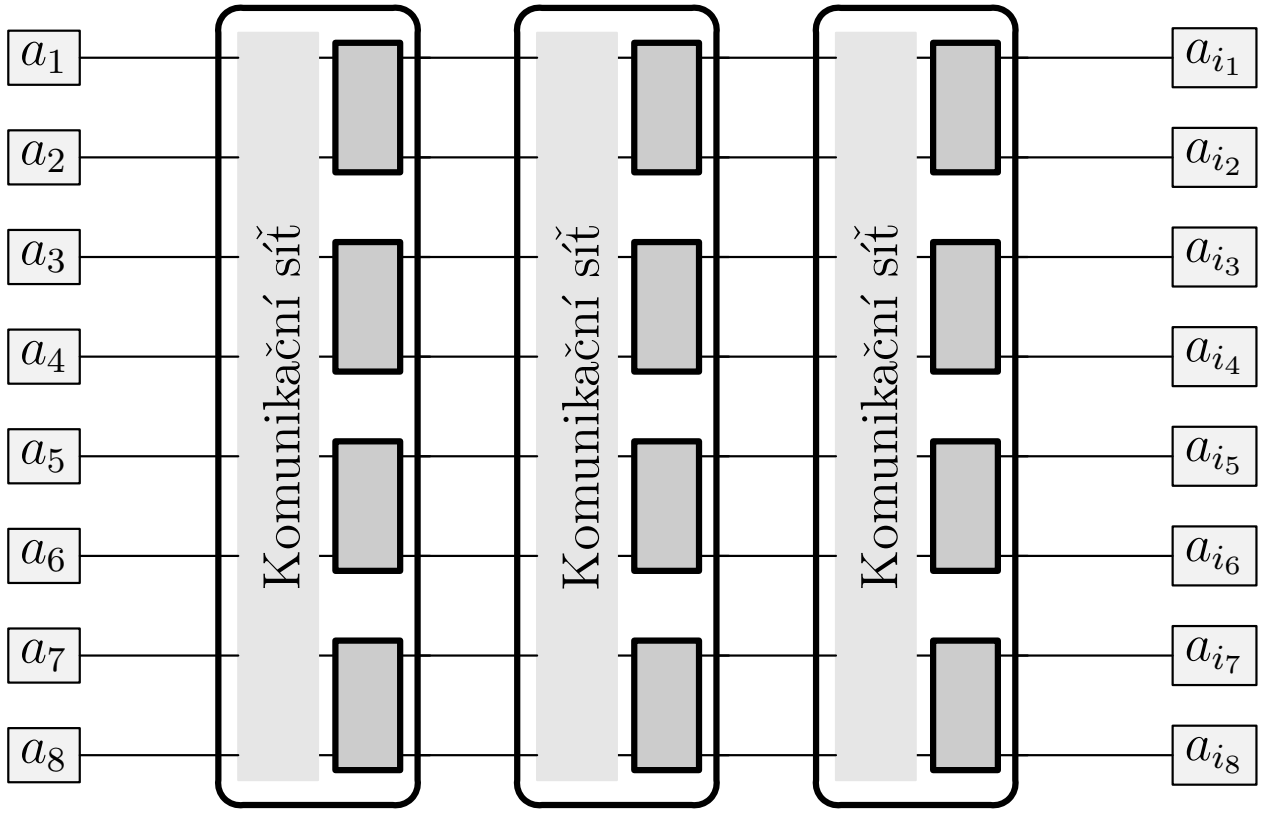
$$x' = \min\{x, y\} \quad y' = \max\{x, y\},$$

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



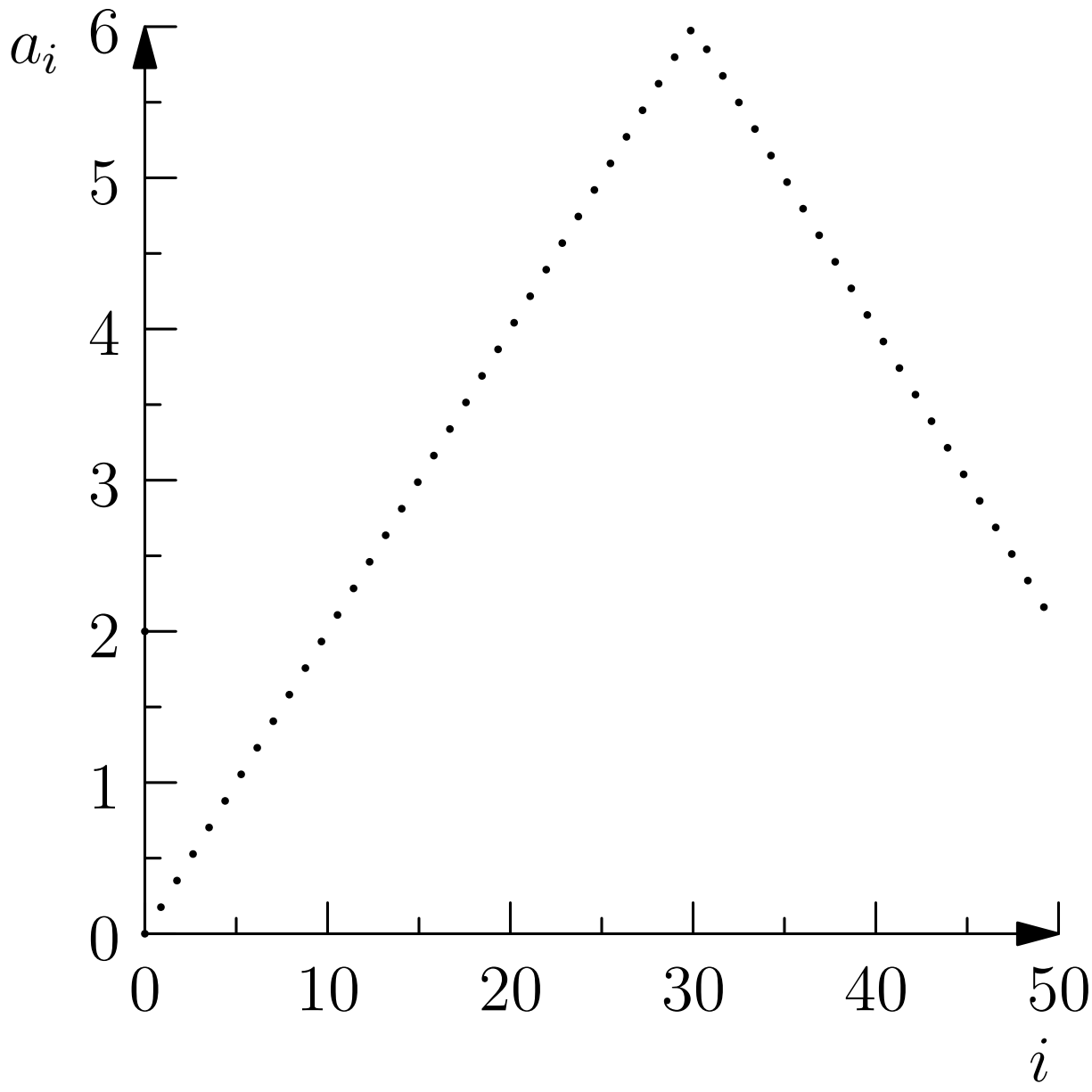
Bitonic sort

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

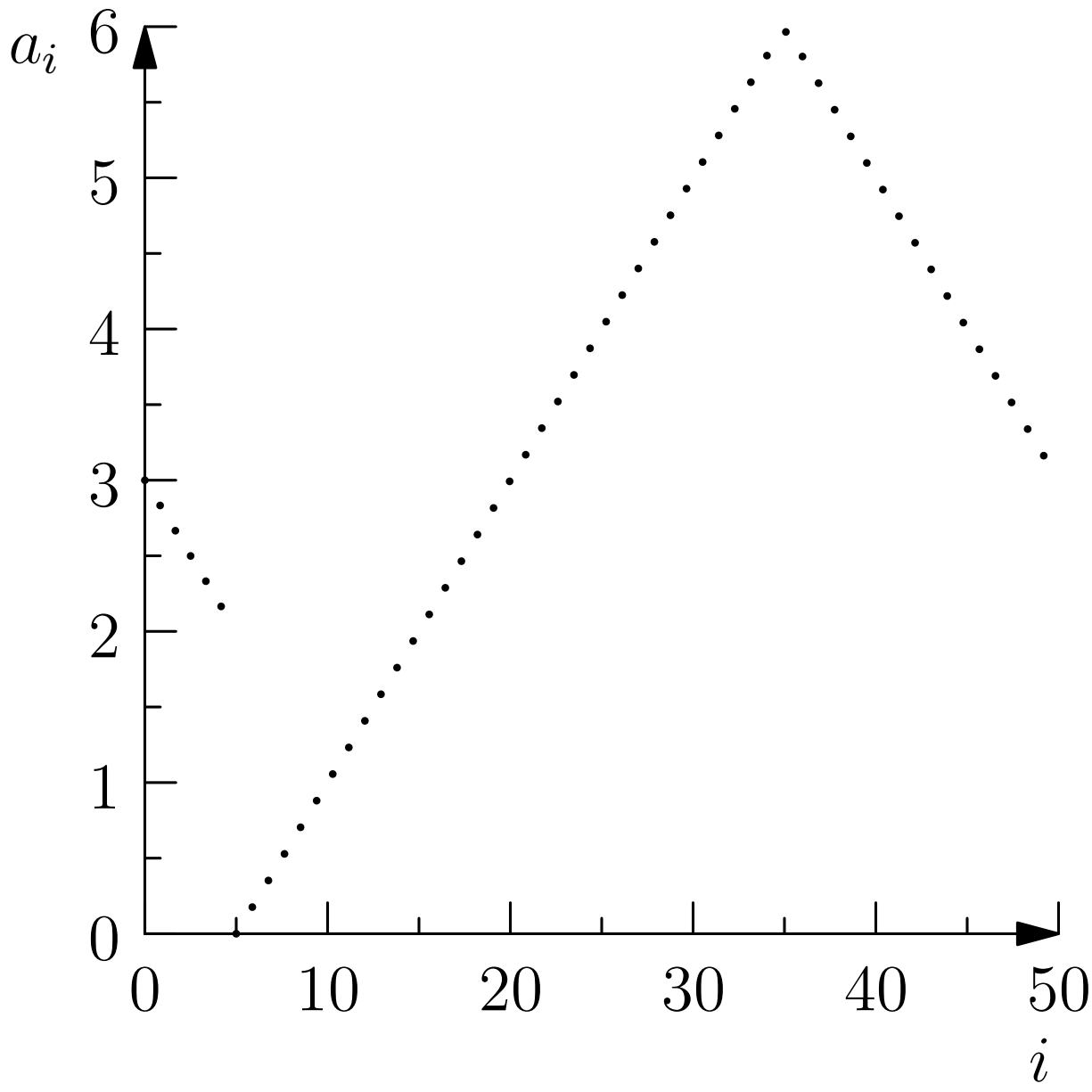
Definition

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

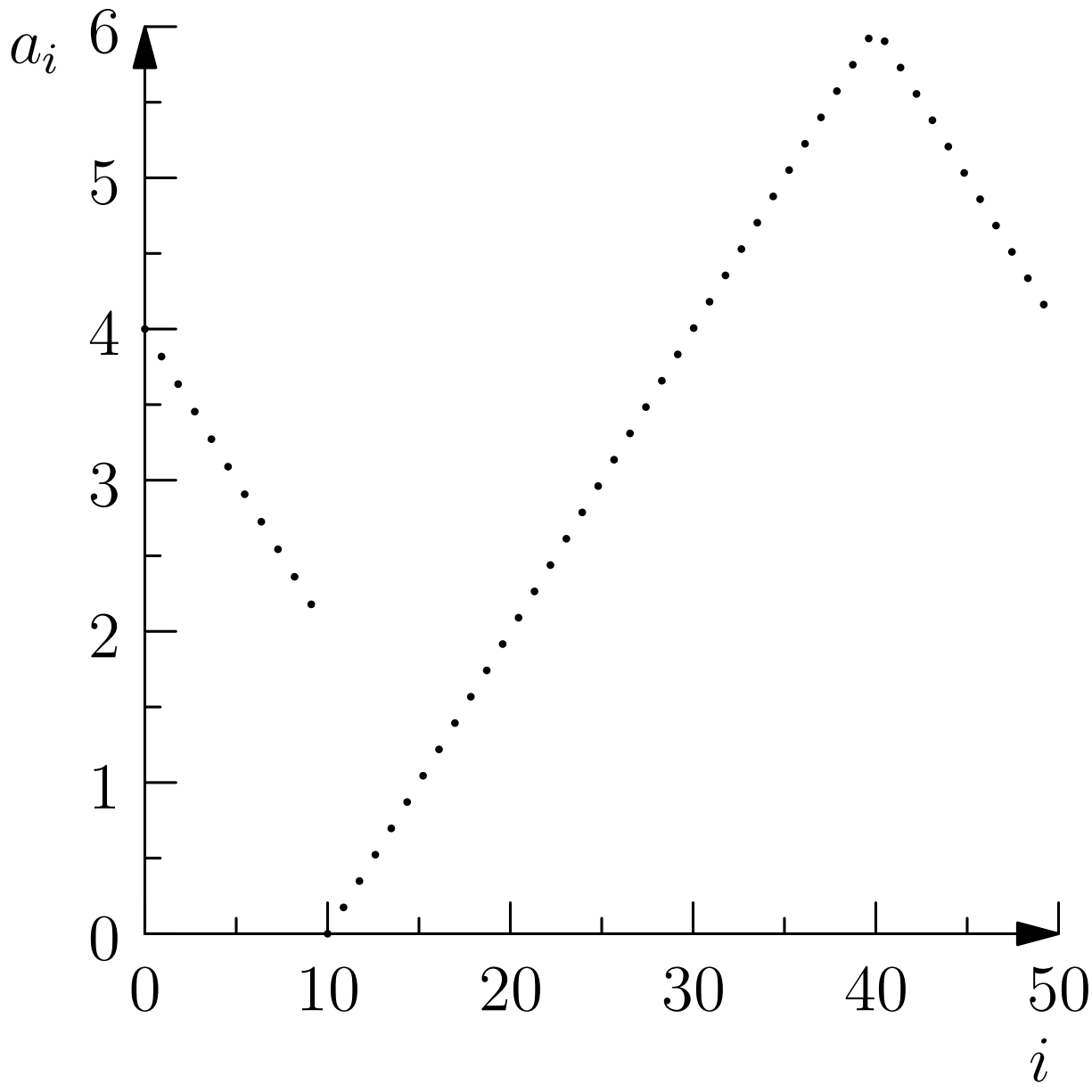
Bitonic sort



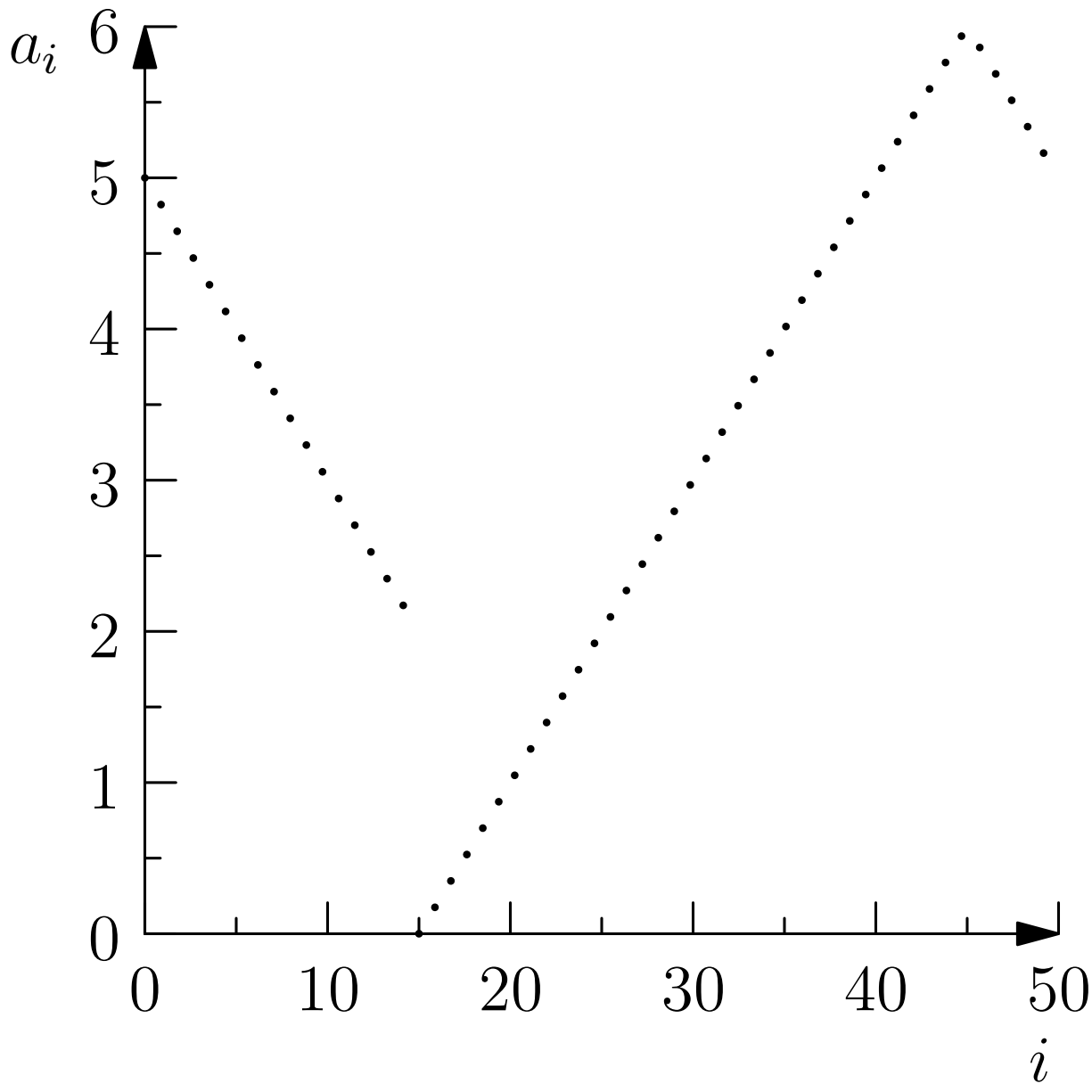
Bitonic sort



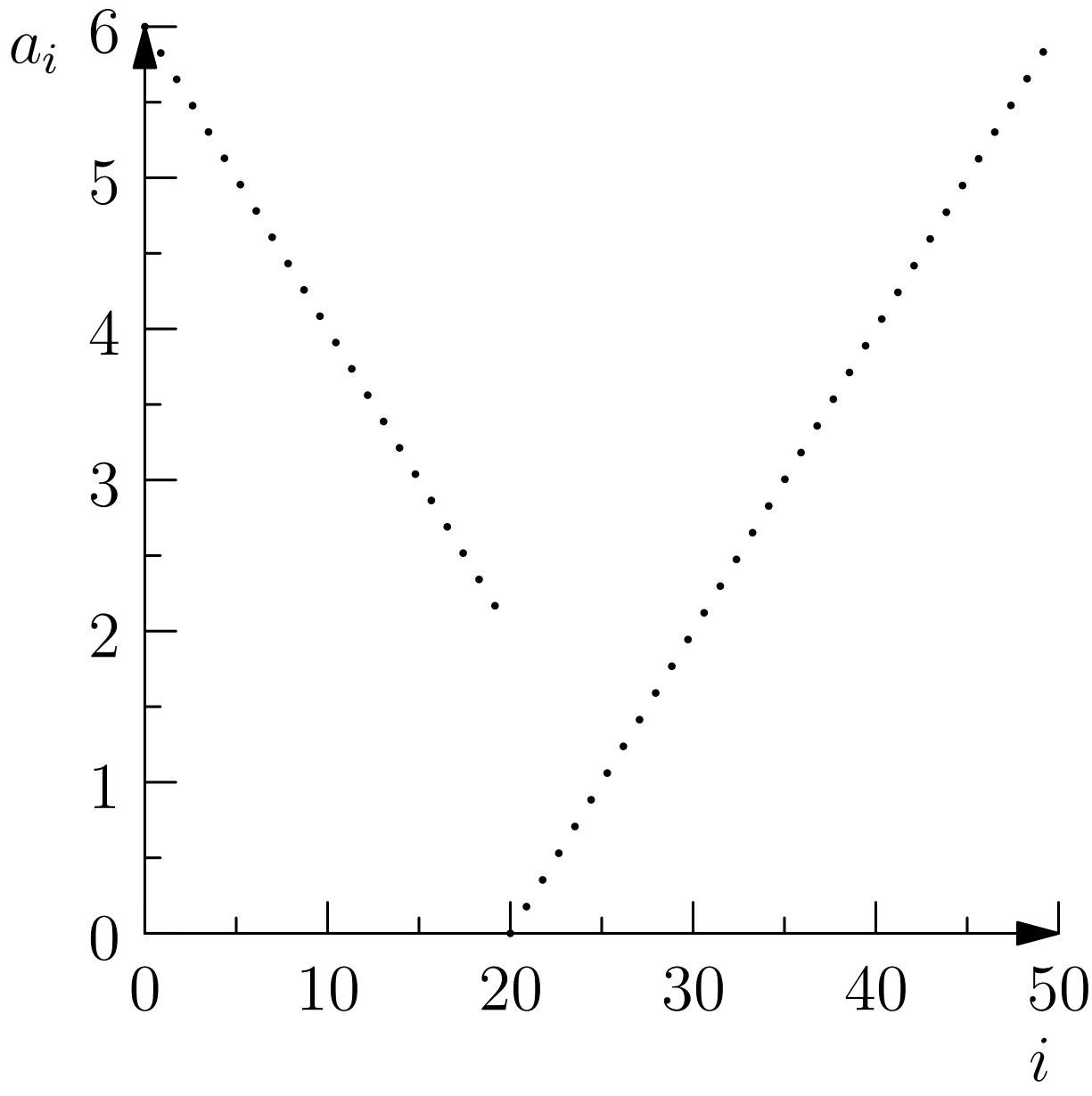
Bitonic sort



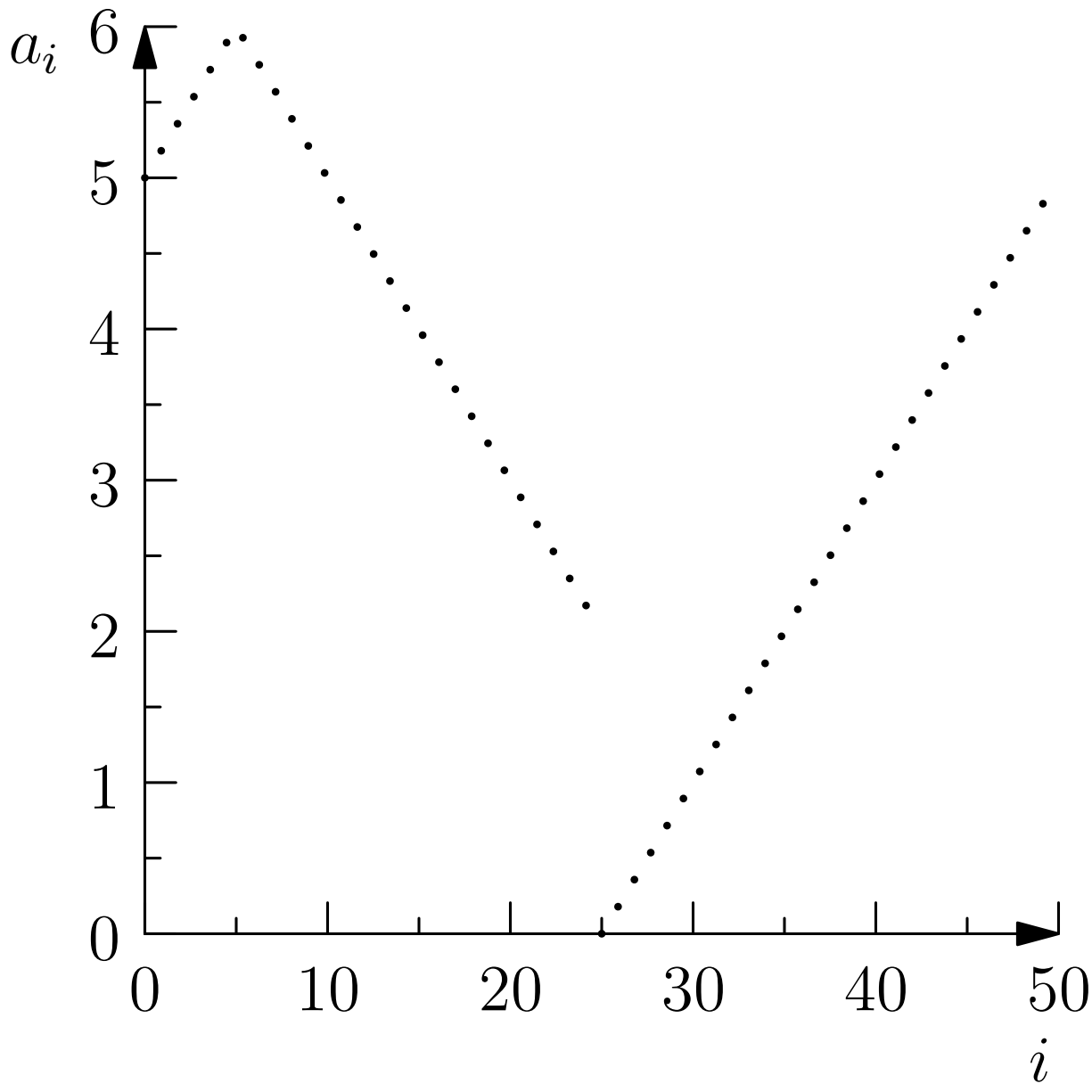
Bitonic sort



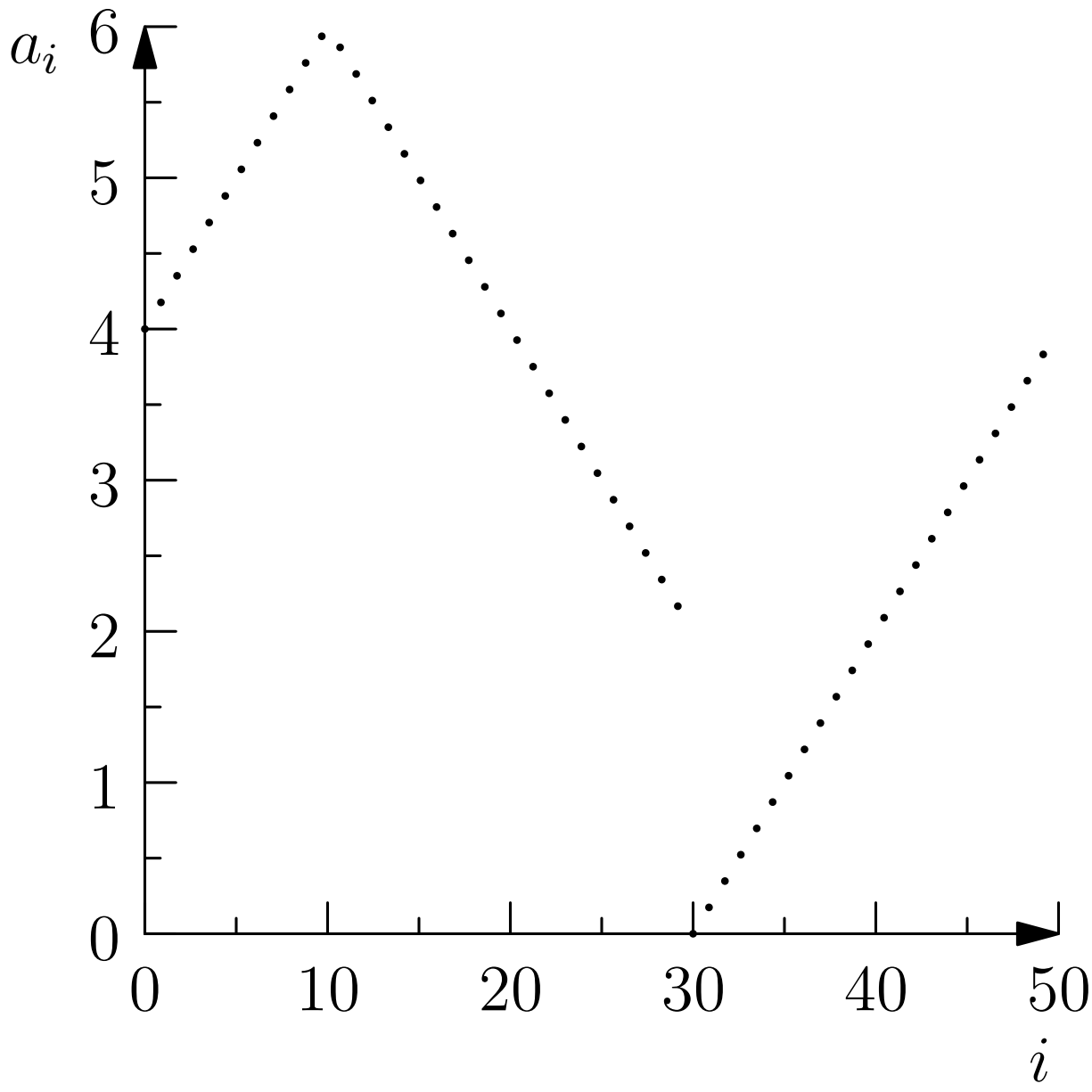
Bitonic sort



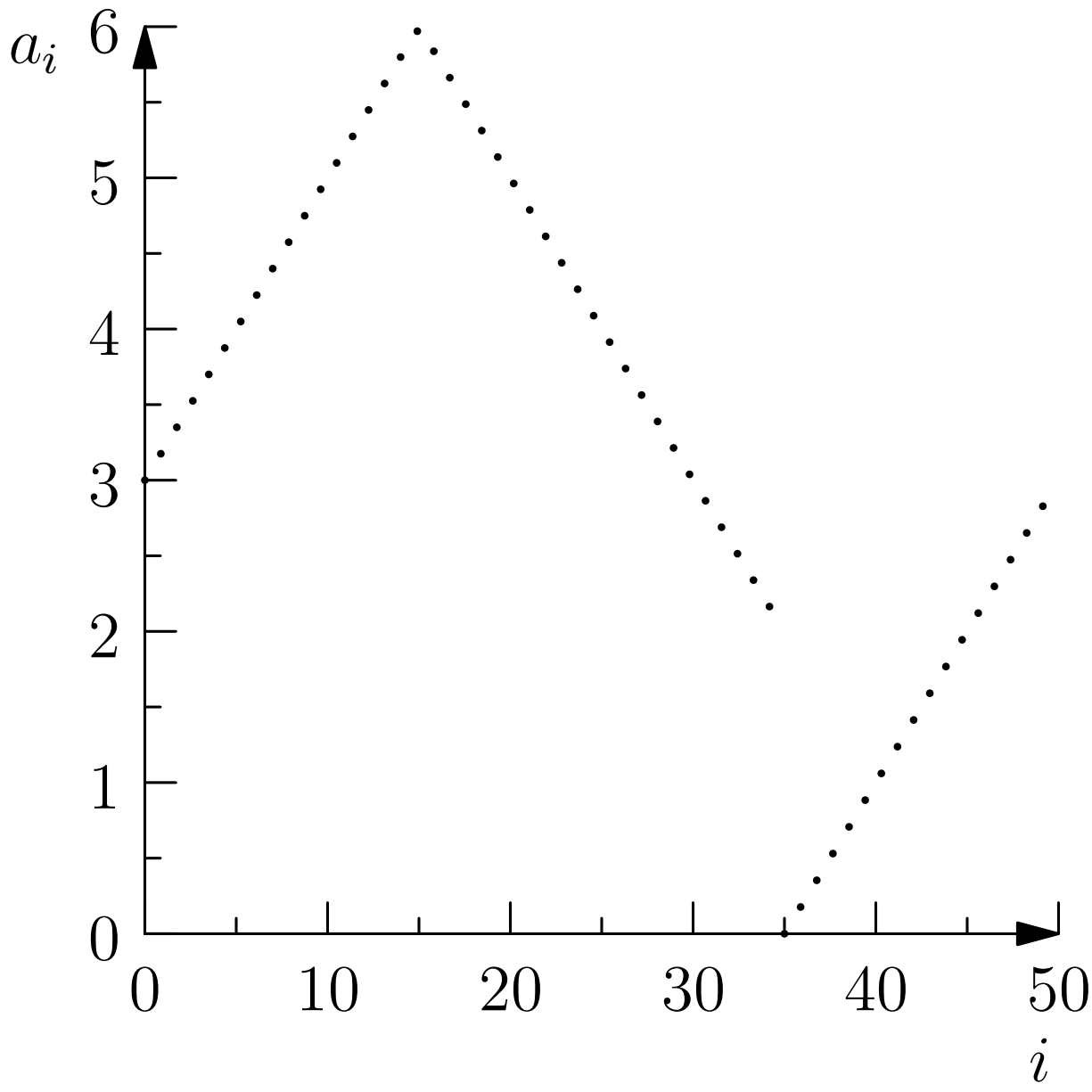
Bitonic sort



Bitonic sort



Bitonic sort

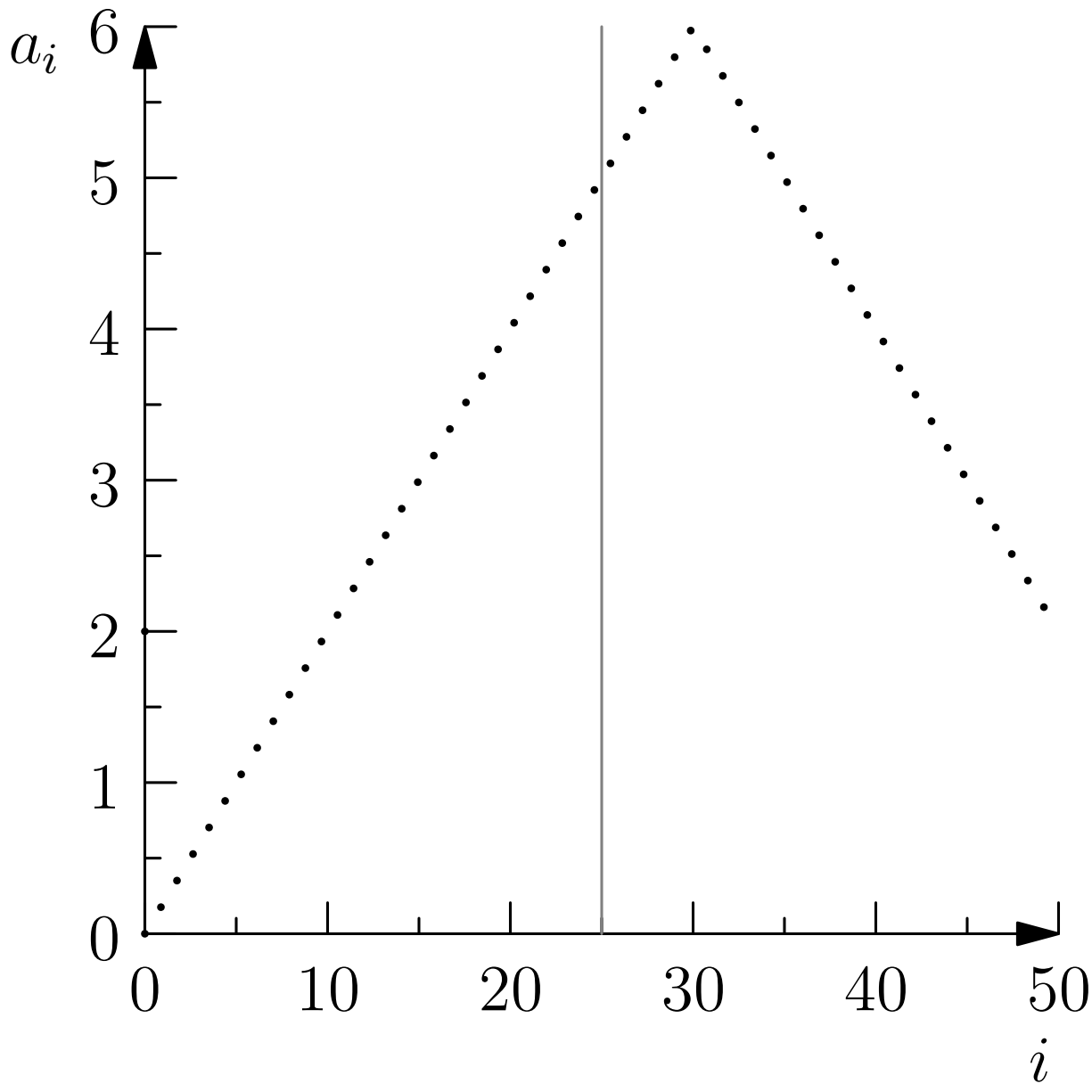


Bitonic sort

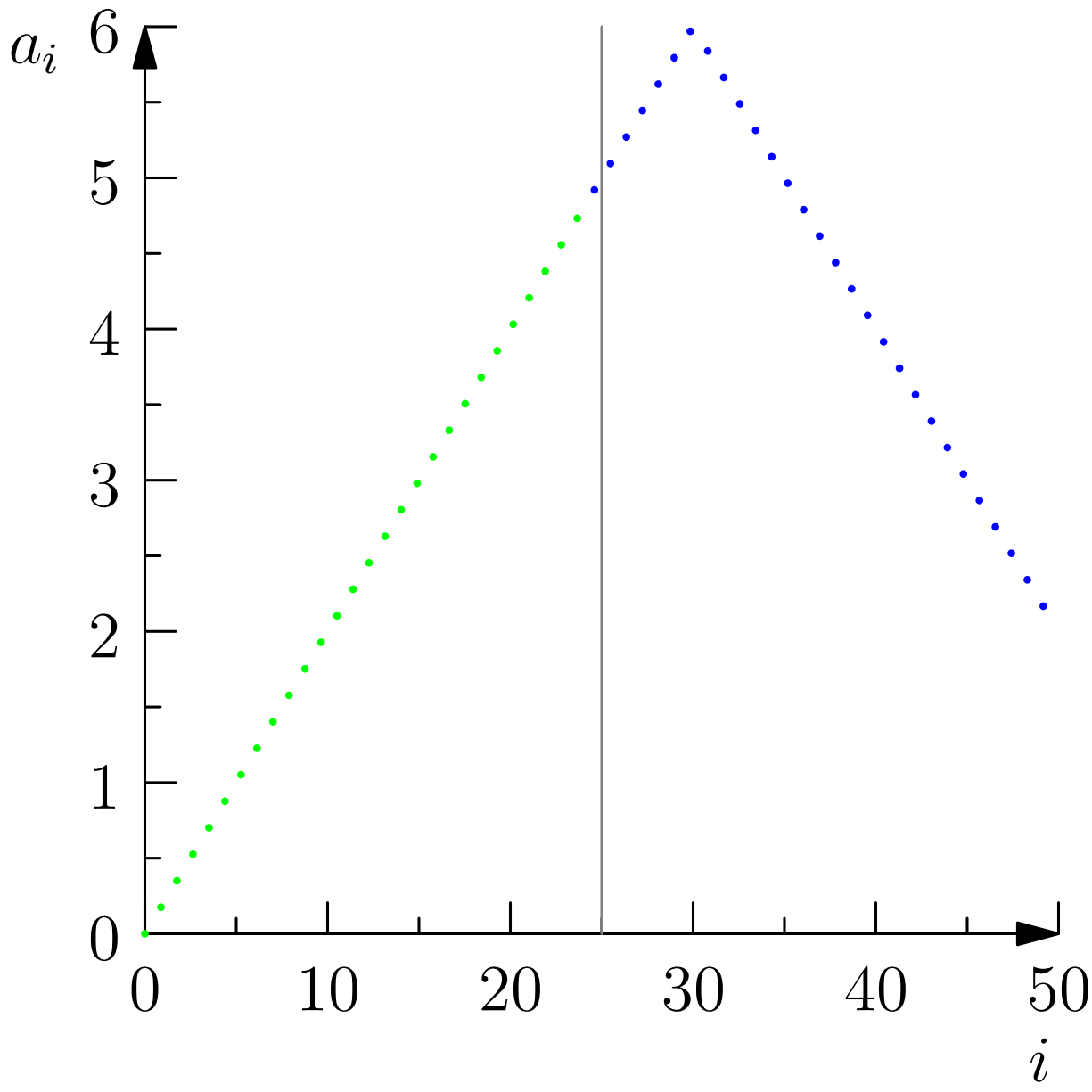
Definujme nyní tyto posloupnosti:

- $$s_1 = \left\{\min\left\{a_0, a_{\frac{n}{2}}\right\}, \min\left\{a_1, a_{\frac{n}{2}+1}\right\}, \ldots, \min\left\{a_{\frac{n}{2}-1}, a_{n-1}\right\}\right\}$$
- $$s_2 = \left\{\max\left\{a_0, a_{\frac{n}{2}}\right\}, \max\left\{a_1, a_{\frac{n}{2}+1}\right\}, \ldots, \max\left\{a_{\frac{n}{2}-1}, a_{n-1}\right\}\right\}$$

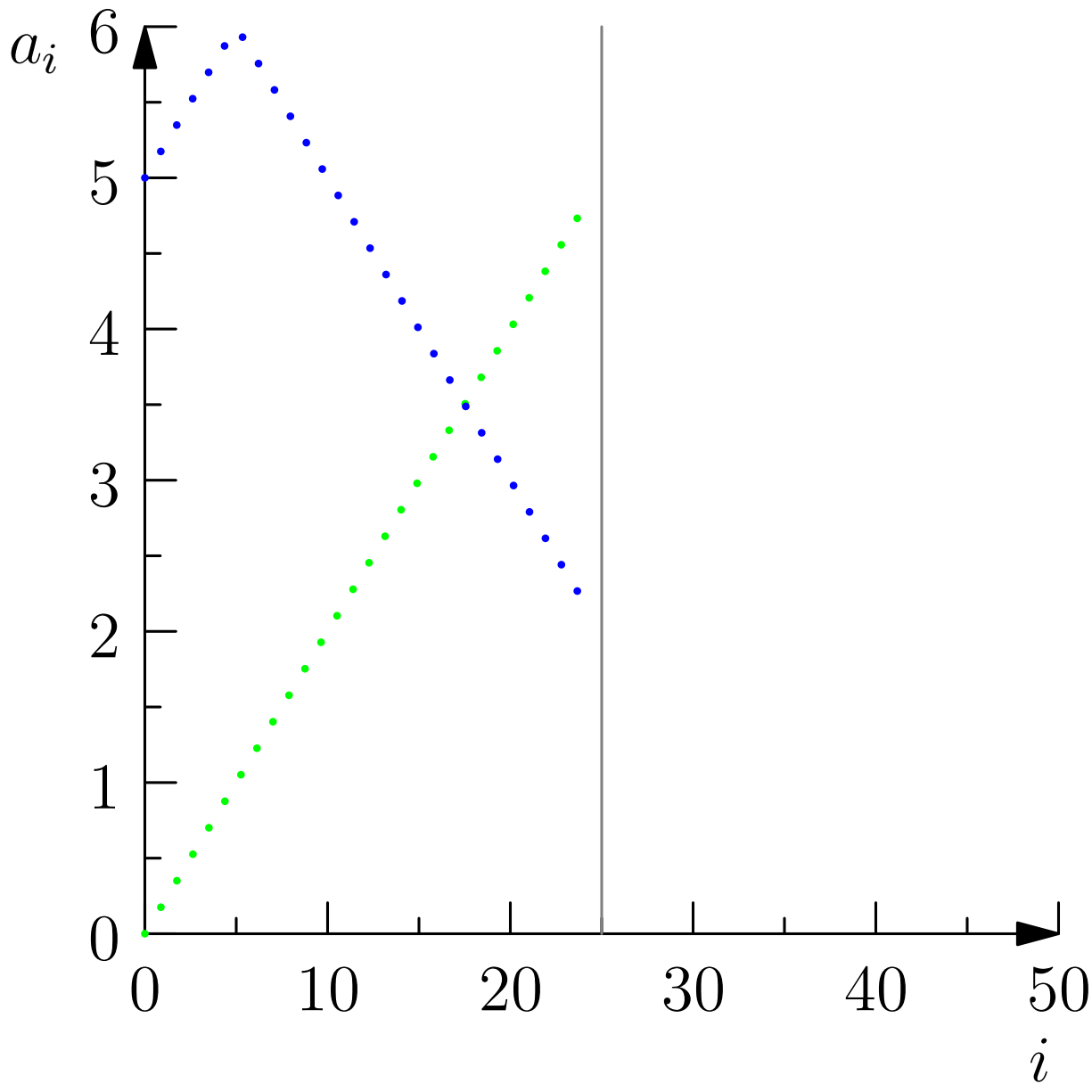
Bitonic sort



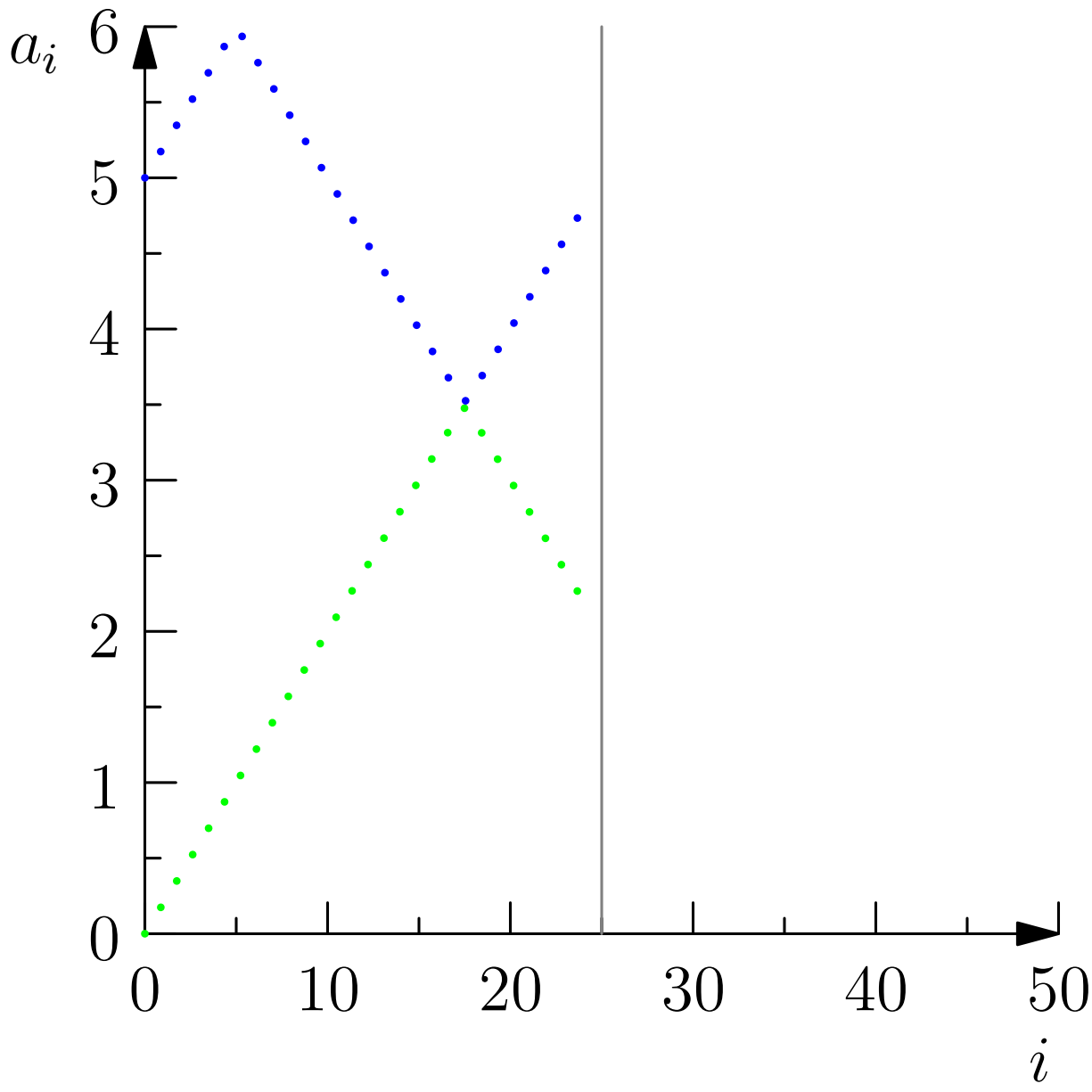
Bitonic sort



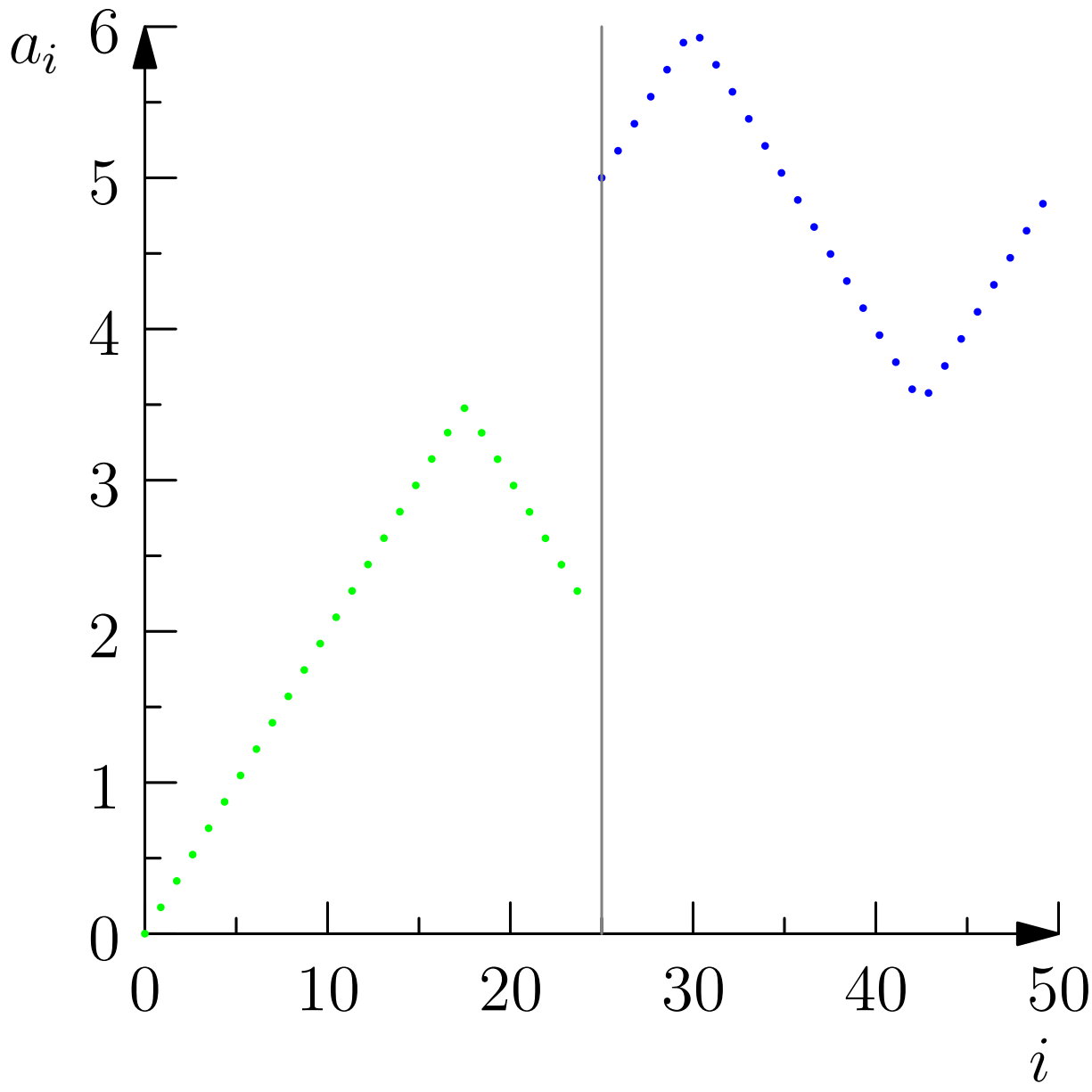
Bitonic sort



Bitonic sort



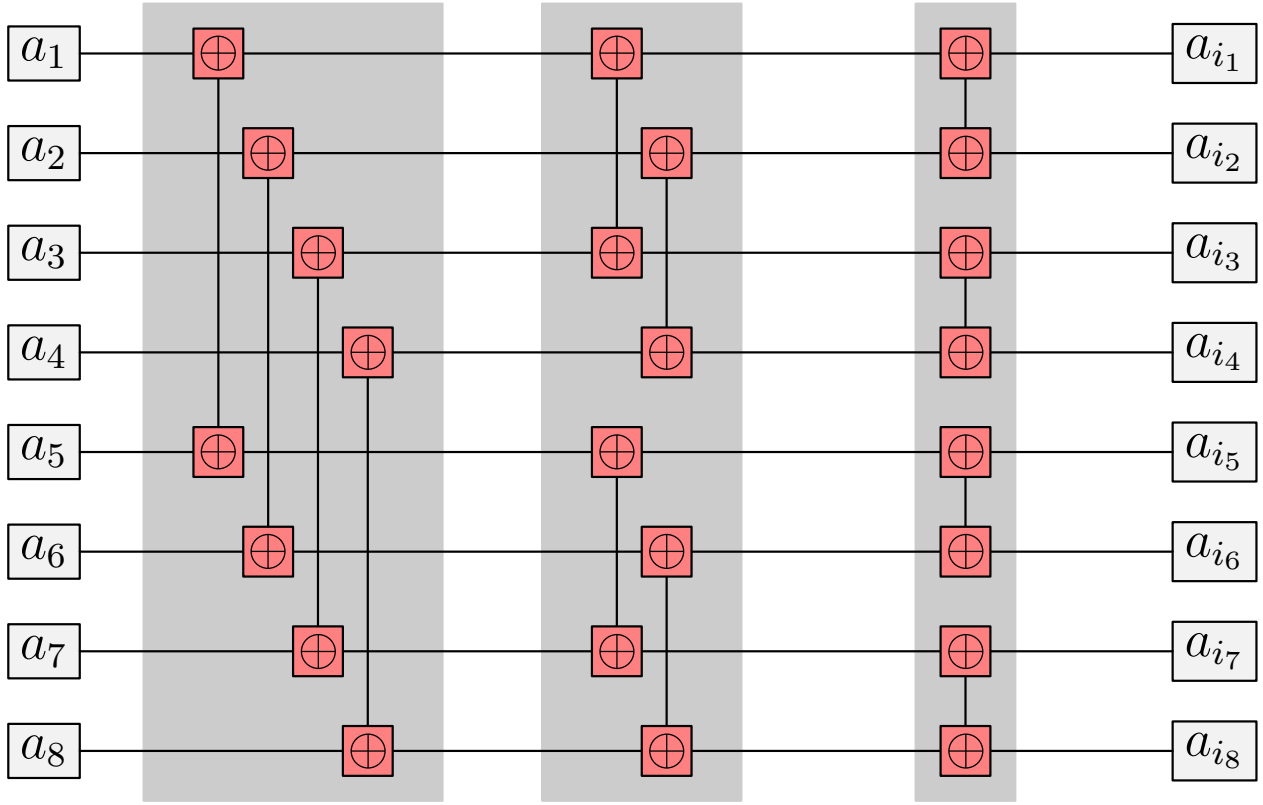
Bitonic sort



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 dojdu k posloupnostem délky 1
- algoritmus nyní zakreslíme pomocí komparátorů

Bitonic sort

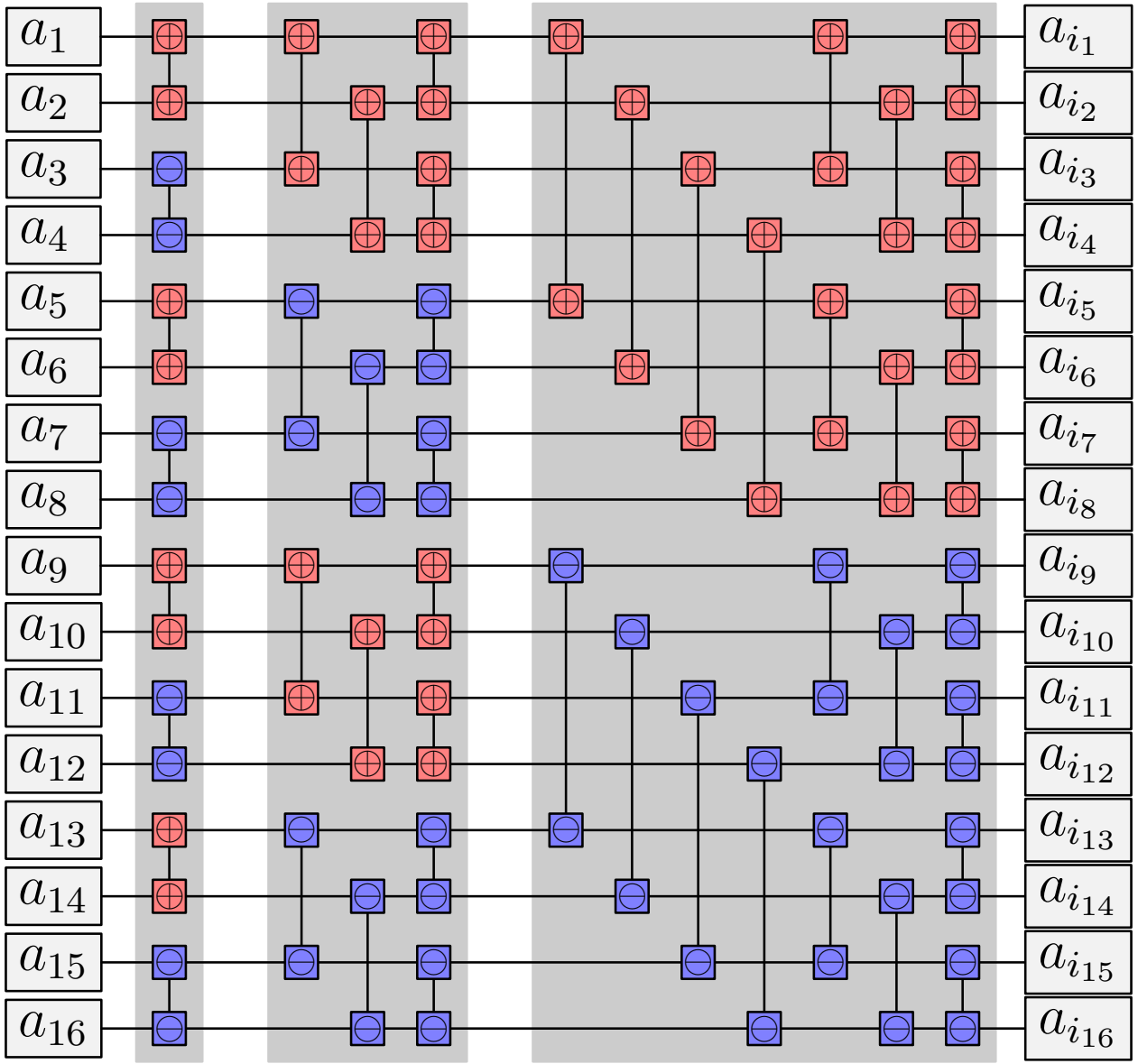


Bitonic sort

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 ukázku si ukážeme, jak získat bitonickou posloupnost o délce 4

Bitonic sort

- máme posloupnost a_1, a_2, a_3, a_4 a víme, že

$$a_1 \leq a_2, a_3 \geq a_4$$

- nová posloupnost vznikne jako

$$a_1^1 = \min\{a_1, a_3\}, a_2^1 = a2, a_3^1 = \max\{a_1, a_3\}, a_4^1 = a_4$$

- dostáváme

$$\begin{aligned} a_1 \leq a_2, a_1^1 = \min\{a_1, a_3\} &\Rightarrow a_1^1 \leq a_2^1, a_1^1 \leq a_3^1, \\ a_3 \geq a_4, a_3^1 = \max\{a_1, a_3\} &\Rightarrow a_3^1 \geq a_4^1. \end{aligned}$$

- dále máme

$$a_1^2 = a_1^1, a_2^2 = \min\left\{a_2^1, a_4^1\right\}, a_3^2 = a_3^1, a_4^2 = \max\left\{a_2^1, a_4^1\right\}.$$

- platí tedy

$$\begin{aligned} a_1^1 \leq a_2^1 \leq \max\left\{a_2^1, a_4^1\right\} = a_4^2 &\Rightarrow a_1^2 \leq a_4^2, \\ a_1^1 \leq a_3^1, a_3^2 = a_3^1 &\Rightarrow a_1^2 \leq a_3^2, \\ a_3^2 = a_3^1 \geq a_4^1 \geq a_2^2 = \min\left\{a_2^1, a_4^1\right\} &\Rightarrow a_2^2 \leq a_3^2, \\ a_2^2 = \min\left\{a_2^1, a_4^1\right\}, a_4^2 = \max\left\{a_2^1, a_4^1\right\} &\Rightarrow a_2^2 \leq a_4^2. \end{aligned}$$

- dostáváme tak, že oba prvky a_1^2 a a_2^2 jsou menší než a_3^2 a a_4^2 a jednoduše už z nich udělám rostoucí posloupnost délky 4

Bitonic sort

Analýza bitonického třídění:

- při vytváření setříděné posloupnosti z bitonické máme snadno pro "hloubku"sítě $D^*(n)$

$$D^*(n) = \sum_{i=1}^{\log n} i = \frac{1}{2}(\log^2 n + \log n)$$

- při vytváření bitonické posloupnosti má každá úroveň i "hloubku"stejnou jako $D^*(i)$

$$D(n) = \sum_{i=1}^{\log n} D^*(i) \approx \theta(\log^2 n)$$

- $T_P(n,p) = n/p \log^2 n$
- $S(n,p) = n/p^2 \log^2 n$
- $E(n,p) = n/p^3 \log^2 n$
- $C(n,p) = n \log^2 n$ tj. není nákladově optimální

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=1;i<=r;i++) aux[count[digit(a[i],w)]++] = a[i];
10     //for(i=l;i<=r;i++) a[i] = aux[i];
11     switch( a, aux );
12     radixMSD( a, l, count[0] + l-1, w+1 );
13     for(j=0;j<R-1;j++)
14        radixMSD( a, count[j] + l-1, count[ j+1 ] + l-1, w+1 );
15
16  }
```

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 localCount[ R+1 ];
11          for(j=0;j<R;j++) localCount[j]=0;                    // !!!
12
13          #pragma omp for private(i) schedule (static)
14          for(i=0;i<n;i++) localCount[ digit( a[i], w ) + 1 ][pid]++; // !!!
15
16          for(j=0;j<R;j++) count[pid][j]=localCount[j]; // !!!
17
18          partialSum(count, localCount);
19
20          #pragma omp for private(i) schedule (static)
21          for(i=0;i<n;i++) aux[localCount[digit(a[i],w)]++] = a[i];
22
23          #pragma omp single
24          swap(a, aux);
25      }
26  }
```

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
- bohužel ta se ale těžko paralelizuje

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 se 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 se 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 se 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 se 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

Radix sort na GPU

N. Satish, M. Harris, M. Garland, *Designing Efficient Sorting Algorithms for Manycore GPUs*, Proc. 23rd IEEE International Parallel and Distributed Processing Symposium, May 2009.

- viděli jsme, že radix sort vyžaduje operaci prefix-sum
- ta je na GPU komplikována hierarchií různě rychlých pamětí
- navíc při zápisu jednotlivých prvků do přihrádek provádíme nesekvenční zápis, který je označován jako scatter
- dnešní GPU umí dobře scatter v rámci warpu
- chceme
  - minimalizovat počet operací scatter do globální paměti
  - maximalizovat koherenci operace scatter
- obojího lze dosáhnout zmenšením velikosti přihrádek
- základ pro radix sort R volíme jako $R = 2^b$

Radix sort na GPU

1. Fáze (vlastní kernel)
   - každý CUDA blok si načte svůj blok dat
     - pro začátek volíme velikost CUDA bloku 256 a stejně tak i datového bloku
   - následně provede setřídění pomocí b 1-bitových splitů
   - výsledek zapíše do globální paměti

Radix sort na GPU

Co je 1-bitový split?

- podívám se na daný *k*-tý bit každého prvku tříděného pole
- je-li tento bit roven 0, zapíšu do pole `Aux1` jedničku, jinak zapíšu jedničku do pomocného pole `Aux2` (obě pole jsou jinak nulová)
- na obou polích provedu prefix-sum
- je-li daný bit nulový, přesunu prvek na pozici *i* na pozici danou `Aux1[i]-1`
- jinak ho zapíšu na pozici `aux1Max + Aux2[i]-1`
- tím zaručím, že neporuším uspořádání podle předchozích bitů v rámci jedné "přihrádky"

Radix sort na GPU

2. Fáze (nový kernel)

- každý blok si načte svoje prvky
- napočítá histogram zastoupení jednotlivých v 2^b hodnot
- výsledek je uložen v matici o rozměrech $p \times 2^b$, kde p je celkový počet bloků
- (i, j)-tý prvek říká, kolik je v i-tém bloku prvků, jejichž bitový zápis tvořený b bity podle nichž zrovna třídíme se rovná j.
- tabulka se zapíše do globální paměti

Radix sort na GPU

3. Fáze (opět nový kernel)
   - provedeme prefix sum v celé tabulce, po sloupcích
   - tím nejprve vysčítáme všechny prvky s hodnotou 0 ve všech blocích apod.
4. fáze (opět nový kernel)
   - podle výsledku prefix sum zapíše každé vlákno svůj prvek na nové místo v paměti

Nejvhodnější volba pro b se zdá být 4. Také je vhodné nechat každé vlákno zpracovávat 4 prvky, pro větší efektivitu.

Enumeration sort

Jde o teoretický algoritmus pro CRCW PRAM.

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

```
procedure enumSort( A, n )
begin
    for each P_1j do C[ j ] := 0;
    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
```