ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Paralelní metody stabilního řazení
Student:	Michal Čermák
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Nastudujte algoritmy InsertionSort, MergeSort (in-place a out-place, viz [1,2]) a jejich paralelní varianty (vícevláknové pod sdílenou pamětí).
2) Implementujte je v jazyce C++ a optimalizujte transformacemi kódu a paralelizací pomocí technologie OpenMP.
3) Porovnejte výkonnost a pamětové nároky in-place a out-place MergeSortu.
4) Navrhněte paralelní hybridní algoritmus využívající kombinaci in-place, out-place MergeSortu a InsertionSortu
s ohledem na maximální výkonnost a množství dodatečné paměti.
5) Změřte výkonnost tohoto hybridního algoritmu na fakultním serveru Star pro různé vstupní posloupnosti a pro různé typy řazených objektů (char, int, string, ...).
6) Porovnejte výkonnost hybridního algoritmu s existujícími implementacemi paralelního stabilního řazení, zejména s [3].

Seznam odborné literatury

[1] Antonios Symvonis: Optimal Stable Merging, The Computer Journal, 1993.
[2] Jyrki Katajainen, Tomi Pasanen, Jukka Teuhola: Practical in-place mergesort . Nordic Journal of Computing, 1996
[3] Arch D. Robison. A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP. Intel® Software. [online]. 11.4.2014 [cit. 2016-11-18]. Dostupné z: https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 4. ledna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Paralelní metody stabilního řazení

Michal Čermák

Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

8. ledna 2018

Poděkování

Děkuji vedoucímu mé bakalářské práce, panu doc. Ing. Ivanu Šimečkovi, Ph.D., za cenné rady a vstřícný přístup při psaní této práce. Dále děkuji své rodině za podporu v průběhu celého studia a zejména během psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. ledna 2018 .....

Odkaz na tuto práci

Čermák, Michal. *Paralelní metody stabilního řazení*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce obsahuje popis algoritmů in-place a out-of-place Mergesortu. Zabývá se jejich implementací v jazyce C++ a následnou optimalizací a paralelizací pomocí technologie OpenMP. Dále je na základě těchto algoritmů vytvořen hybridní algoritmus využívající principů in-place a out-of-place Mergesortu s ohledem na efektivní využití poskytnutého pomocného pole různých délek. Na závěr je výkonnost tohoto hybridního algoritmu porovnána s výkonností několika vybraných existujících implementací stabilního řazení.

Klíčová slova Paralelní řazení, Mergesort, Insertion sort, Optimalizace, Inplace Mergesort

Abstract

This thesis contains a description of in-place and out-of-place Mergesort algorithms. It describes their implementation in the C++ language followed by their optimization and parallelization using the OpenMP technology. Furthermore, a hybrid algorithm based on these algorithms is created with the aim of an effective usage of a provided buffer of various lengths. Lastly, the performance of this hybrid algorithm is compared with the performance of several selected existing implementations of stable sorting.

Keywords Parallel sorting, Mergesort, Insertion sort, Optimization, In-place Mergesort

Obsah

Seznam algoritmů

Seznam obrázků

Seznam tabulek

Úvod

S nějakou formou řazení se v našich životech nejspíše setkala většina z nás, ať už se jednalo o seřazení různých dokumentů podle abecedy nebo nějakých objektů podle velikosti či váhy. Řazení je pro nás intuitivní záležitostí a často ani nepřemýšlíme nad tím, jak ho vlastně provádíme. Proč jsme daný předmět umístili právě na toto místo. Právě způsob jak řadit, tedy jeho algoritmizace, je v informatice klíčová, stejně jako v této práci.

V informatice je řazení velmi známým problémem, pro který bylo vymyšleno mnoho algoritmů. Většina těchto algoritmů má stále svá využití, ať už se jedná o nejvhodnější volbu v nějaké situaci nebo jako příklad k naučným účelům. Ačkoliv se jedná o problematiku zkoumanou už celou řadu let, stále se v ní najdou problémy, které lze zkoumat dále.

Řazení má celou řadu využití. Některé algoritmy využívají řazení, aby samy mohly být efektivnější. Ať už se jedná o větší efektivitu z hlediska asymptotické složitosti nebo za účelem lepšího využití cache pamětí, jako např. v některých případech násobení matic.

V dnešní době, kdy místo zvyšování frekvence procesoru dochází ke zvyšování počtu jader, je důležitá pro výkonnost algoritmů efektivní paralelizace. A právě paralelizace algoritmů řazení je ten problém, kde je stále co zkoumat. Proto se tato práce zabývá paralelními metodami řazení.

Kapitola 1

Cíle práce a základní pojmy

Tato kapitola obsahuje vysvětlení základních pojmů užívaných v této práci. Také obsahuje definici cílů této práce.

1.1 Cíle práce

Cílem této práce je implementace algoritmů stabilního řazení, následně jejich optimalizace a paralelizace. Konkrétně se jedná o algoritmus *Mergesort*, jeho *out-of-place* a *in-place* varianty. Cílem této práce je také srovnání časové a paměťové náročnosti těchto dvou variant. Dalším cílem je vytvoření hybridního algoritmu kombinujícího obě varianty *Mergesortu*, změření jeho výkonnosti a porovnání s existujícími implementacemi paralelního stabilního řazení.

1.2 Řazení

Problém řazení je problémem nalezení permutace P vstupní posloupnosti A o $n \in N$ prvcích takové, že platí:

$$\forall i, j \in N : 1 \leq i < j \leq n \implies P_i \leq P_j$$

kde P_x vyjadřuje x-tý prvek v permutaci P.

Pro řazení existuje mnoho algoritmů. Mezi ty nejznámější patří *Insertion sort*, *Quicksort* a *Mergesort*. Tato práce se zabývá algoritmy *Insertion sort* a *Mergesort*, které jsou podrobněji popsány v rámci kapitoly 2.

1.2.1 Klasifikace algoritmů řazení

Algoritmy řazení lze dělit podle několika kritérií.

1.2.1.1 Stabilita

Řazení je stabilní, pokud zachovává pořadí prvků, které jsou si rovny, tedy platí:

$$\forall i, j \in N, 1 \leq i < j \leq n : A_i = A_j \implies P(A_i) < P(A_j)$$

kde A_x označuje x-tý prvek ve vstupní posloupnosti A a $P(A_x)$ pozici tohoto prvku v seřazené permutaci P. Pokud tato podmínka neplatí, jedná se o řazení nestabilní.

1.2.1.2 Adaptivita

Algoritmus je adaptivní, pokud dokáže využít již seřazených podposloupností. Jednoduchým příkladem adaptivního algoritmu řazení je jakýkoliv, který nejprve provede kontrolu, zda již vstupní posloupnost je seřazená, a pokud ano, ukončí se.

1.2.1.3 Základní operace

Podle užívaných operací se algoritmy dělí na ty, které využívají pouze operace porovnání a prohození prvků, a na ostatní. Algoritmy, které používají jiné operace, nejsou obecně použitelné na všechny druhy vstupů. Toto rozdělení je důležité pro určení dolního odhadu časové složitosti a tedy určení, zda je algoritmus asymptoticky optimální.

1.2.1.4 Časová složitost

Zřejmá dolní hranice pro vstup délky n je $\mathcal{O}(n)$, protože i v nejlepším případě se musí pro každý prvek provést kontrola, zda se nachází na správném místě. Nicméně optimální časová složitost pro algoritmy používající pouze porovnání a prohození prvků, které jsou zaměřením této práce, je $\mathcal{O}(n \log n)$, což je dokázáno v [1].

1.2.1.5 Paměťová složitost

Algoritmy se dále dělí podle množství využité dodatečné paměti. Podle [2], pokud nepotřebují více než $\mathcal{O}(\log n)$ paměti, jedná se o *in-place* algoritmy. V opačném případě jde o *out-of-place* algoritmy.

1.3 Zrychlení

Jako zrychlení chápeme poměr dosaženého času jedním algoritmem ku času dosaženému jiným algoritmem, se kterým se snažíme daný algoritmus porovnat. Pokud je tento poměr větší než 1, je tedy tento algoritmus rychlejší, uvádíme, že došlo ke zrychlení. V opačném případě říkáme, že ke zrychlení nedošlo, nebo že došlo ke zpomalení.

Pokud měříme zrychlení paralelního algoritmu, chápeme tím zrychlení vůči času dosaženého sekvenční verzí daného algoritmu.

1.4 Cache paměť

Informace o cache paměti jsou čerpány z [3, 4]. Cache paměť je paměť umístěna mezi procesorem a hlavní pamětí. Její velikost je řádově nižší, ale zároveň také její přístupová doba je výrazně nižší než u hlavní paměti. Slouží k uchovávání vybraných dat z hlavní paměti, aby mohl přístup k těmto datům být rychlejší. Pro výběr těchto dat využívá časovou a prostorovou lokalitu dat.

- Časová lokalita – Pokud byla použita nějaká data, budou pravděpodobně v nejbližší době použita znovu.
- Prostorová lokalita – Pokud byla použita nějaká data, budou pravděpodobně použita i jiná data v blízkosti použitých dat.

Efektivní využití cache paměti vzhledem k těmto principům je důležité při optimalizaci algoritmů.

Kapitola 2

Základní algoritmy

Tato kapitola obsahuje popis základních algoritmů používaných v této práci.

2.1 Insertion sort

Informace o tomto algoritmu jsou čerpány z [5]. Insertion sort je algoritmus založený na porovnání a prohození prvků. Je stabilní a adaptivní, jeho asymptotická časová složitost je však $\mathcal{O}(n^2)$, což není optimální. Nicméně je díky své jednoduchosti pro krátké vstupy velmi efektivní a je tedy vhodný pro ukončení rekurze u algoritmů, jako je *Mergesort*. Algoritmus 2.1 je ukázkou pseudokódu pro Insertion sort.

Algoritmus 2.1 Insertion sort

```
1:  procedure InsertionSort(array, begin, end)
2:      for i ← begin + 1 to end do
3:          j ← i
4:          while j > begin and array[j] < array[j − 1] do
5:              tmp ← array[j]
6:              array[j] ← array[j − 1]
7:              array[j − 1] ← tmp
8:              j ← j − 1
9:          end while
10:     end for
11: end procedure
```

2.2 Mergesort

Základní informace o tomto algoritmu jsou čerpány z [6]. *Mergesort* je algoritmus stabilního řazení. Byl vyvinut Johnem von Neumannem v roce 1945. Řadí

se mezi algoritmy typu rozděl a panuj. Rekurzivně se nejprve zavolá na první a druhou polovinu vstupní posloupnosti a poté spojí obě seřazené poloviny v jednu. Výsledkem je seřazená posloupnost. Algoritmus není adaptivní, jeho časová složitost je $\mathcal{O}(n \log n)$ a paměťová náročnost závisí na funkci *Merge*, která je typicky *out-of-place* a využívá $\mathcal{O}(n)$ paměti navíc. Algoritmus 2.2 je příkladem pseudokódu pro funkci Mergesort.

Algoritmus 2.2 Mergesort

```
1: procedure MergeSort(array, begin, end)
2:     if begin < end then
3:         half ← ⌊(begin + end)/2⌋
4:         MergeSort(array, begin, half)
5:         MergeSort(array, half + 1, end)
6:         Merge(array, begin, half, end)
7:     end if
8: end procedure
```

2.2.1 Merge

Funkce Merge je nejdůležitější částí Mergesortu. Jelikož je Mergesort stabilní algoritmus, musí funkce Merge zachovávat pořadí prvků, které jsou si rovny.

Existuje mnoho možností, jak dvě seřazené posloupnosti spojit v jednu seřazenou posloupnost. Rozdělují se na dva základní typy. Funkce, které používají pro sloučení pomocné pole – *out-of-place merge* – a funkce, které pomocné pole nepotřebují – *in-place merge*.

Algoritmus 2.3 je ukázkou Merge s pomocným polem. Porovnají se první prvky obou posloupností a menší z nich se přesune do pomocného pole. Pokud jsou si prvky rovny, je přesunut prvek z levé posloupnosti, což zajistí stabilitu řazení. Poté, co byly všechny prvky seřazeny do pomocného pole, jsou přesunuty zpět do pole původního.

2.3 Mergesort s polovičním pomocným polem

Informace o tomto algoritmu jsou čerpány z [7]. Tato varianta Mergesortu je založena na metodě slučování polí využívané v mnoha *in-place* variantách. Pro sloučení posloupností u a v, kde $|u| \leq |v|$, je postačující pomocné pole délky $|u|$. Kratší z posloupností se přesune do pomocného pole a obě posloupnosti se pak sloučí do původního umístění posloupnosti u. Je zřejmé, že pro přesunutí prvků z pomocného pole bude vždy dostatečné volné místo, protože jeho délka je vždy rovná $|u|$, jelikož při obsazení místa prvkem z v se uvolní nové místo na konci.

Při využití tohoto způsobu slučování se oproti klasické variantě sníží délka pomocného pole na $\lfloor \frac{n}{2} \rfloor$, což by mohlo vést k lepšímu využití datové cache

Algoritmus 2.3 Merge

```
1:  procedure MERGE(array, begin, half, end)
2:      buffer ← newArray[begin..end]
3:      rightbegin ← half + 1
4:      bufferindex ← begin
5:      while begin ≤ half and rightbegin ≤ end do
6:          if array[begin] ≤ array[rightbegin] then
7:              buffer[bufferindex] ← array[begin]
8:              bufferindex ← bufferindex + 1
9:              begin ← begin + 1
10:         else
11:             buffer[bufferindex] ← array[rightbegin]
12:             bufferindex ← bufferindex + 1
13:             rightbegin ← rightbegin + 1
14:         end if
15:     end while
16:     while begin ≤ half do
17:         buffer[bufferindex] ← array[begin]
18:         bufferindex ← bufferindex + 1
19:         begin ← begin + 1
20:     end while
21:     while rightbegin ≤ end do
22:         buffer[bufferindex] ← array[rightbegin]
23:         bufferindex ← bufferindex + 1
24:         rightbegin ← rightbegin + 1
25:     end while
26:     array[begin : end] ← buffer[begin : end]
27: end procedure
```

paměti, protože celková velikost využívané paměti bude o čtvrtinu menší. Další výhodou je pak ušetření případných přesunů zbylých prvků pravé poloviny, které už se nacházejí na správném místě. Nevýhodou je pak právě přesun první poloviny před samotným slučováním, což je hlavním zaměřením optimalizací této varianty.

2.4 Operace s bloky prvků

Mergesort bez externího pomocného pole často provádí přesuny skupin prvků, které nemusí být podmíněné jejich porovnáváním. Pro efektivnější manipulaci s nimi je tedy vhodnější pohlížet na ně jako na celé bloky než na jednotlivé prvky. S těmito bloky lze pak provádět několik elementárních operací, mezi které patří např. záměna bloků a rotace bloků.

2.4.1 Záměna bloků

Záměna bloků je jednoduchá záměna prvků ne nutně navazujících bloků u a v, kde $|u| = |v|$. To lze provést s časovou složitostí $\mathcal{O}(|u|)$ jednoduchou záměnou odpovídajících prvků v jednotlivých blocích.

2.4.2 Rotace bloků

Často využívanou operací je rotace dvou navazujících bloků. Tato operace změní pořadí prvků tak, že první prvek pravého bloku bude na místě prvního prvku levého bloku, poslední prvek levého bloku bude na místě posledního prvku pravého bloku a vzdálenosti mezi prvky v obou blocích zůstanou nezměněné. Jinými slovy změní pořadí navazujících bloků uv na blok vu. Jednoduchým pseudokódem funkce, která provede rotaci bloků je algoritmus 2.4. Z pseudokódu je patrné, že časová složitost této operace je $\mathcal{O}(|u|+|v|)$. Funkce Reverse provede obrácení pořadí prvků v daném intervalu.

Algoritmus 2.4 RotateBlocks

```
1: procedure RotateBlocks(array, begin, middle, end)
2:     Reverse(array, begin, middle)
3:     Reverse(array, middle, end)
4:     Reverse(array, begin, end)
5: end procedure
```

2.5 RecMerge

RecMerge je algoritmus uvedený v [8]. Pro účely této práce byl zjednodušen na následující postup. Zvolí se dva pivoty, první z nich bude prvek v polovině levého bloku. Druhý se nalezne pomocí binárního vyhledávání v druhém bloku jako poslední prvek, který je menší než první pivot. Následně se provede rotace bloku od prvního pivota po konec levého bloku s blokem od začátku pravého bloku po druhého pivota. Po této operaci se první pivot nachází ve finální pozici v rámci aktuálního volání funkce Merge. Nalevo i napravo od něj se nachází dva bloky, pro které se rekurzivně zavolá funkce Merge. Rozdílem oproti algoritmu v [8] je, že vždy dochází k půlení levé části pole, namísto dělení delší z částí.

Rekurzivní strom této funkce má hloubku $\mathcal{O}(\log n)$, protože dochází vždy k půlení levé části pole. Tento Merge tedy využívá $\mathcal{O}(\log n)$ paměti na zásobníku. V každé úrovni rekurze se provede $\mathcal{O}(n)$ operací, celková časová složitost je tedy $\mathcal{O}(n \log n)$. Mergesort při použití této varianty funkce Merge má složitost $\mathcal{O}(n \log^2 n)$, což není asymptoticky optimální.

2.6 Straightforward in-place Mergesort

Tento algoritmus, včetně důkazu jeho asymptotické složitosti, je podrobně popsán v [9]. Funguje následujícím způsobem. Nejprve seřadí druhou polovinu pole, k čemuž využije první polovinu jako pomocné pole. Tento princip je popsán v následující sekci 2.6.1. Poté první část postupně dělí na poloviny, seřadí je s druhou polovinou použitou jako pomocné pole a sloučí s již seřazenou částí původního pole opět s využitím druhé poloviny jako pomocného pole. Nakonec z první části zůstane pouze jeden prvek, který je vložen do celého pole lineárně.

Je zřejmé že tento algoritmus není stabilní, protože pořadí identických prvků ve vnitřním pomocném poli nezůstane zachované a nelze ho ani nijak obnovit bez použití $\mathcal{O}(n)$ pomocné paměti. Aby se tento algoritmus stal stabilním, bylo by zapotřebí extrahovat vnitřní pomocné pole o délce $\frac{n}{2}$, což by vedlo ke zvýšení asymptotické časové složitosti algoritmu na $\mathcal{O}(n^2)$, jak bude popsáno dále v sekci 2.7.1, a navíc by to bylo v některých případech nemožné. Tento algoritmus je uveden z důvodu jeho jednoduchosti a k optimalizaci slučování s vnitřním pomocným polem.

2.6.1 Merge s vnitřním pomocným polem

Merge využívající část řazeného pole jako pomocné pole funguje obdobně jako Merge s vnějším pomocným polem, pouze místo přesunu prvků na nové místo se prvky zaměňují, aby nedošlo ke ztrátě dat. Využívá se toho, že pro sloučení posloupností délek u a v, kde $u \leq v$, je postačující pomocné pole délky u, jak bylo popsáno v sekci 2.3.

2.7 Block Mergesort

Tento algoritmus, včetně důkazu jeho asymptotické složitosti, je podrobně popsán v [10]. Přesněji je v [10] popsán algoritmus pro sloučení 2 seřazených posloupnosti, který lze použít v rámci Mergesortu a získat tak stabilní *in-place* řadící algoritmus s asymptotickou časovou složitostí $\mathcal{O}(n \log n)$.

2.7.1 Extrakce rozdílných prvků

Tento algoritmus lze nalézt v [11]. V rámci algoritmu *Block Mergesort* se používá ke slučování tzv. *vnitřní pomocné pole*. Jelikož v průběhu slučování dochází k záměně pořadí prvků v tomto pomocném poli, je pro stabilitu nutné zajistit, aby obsahovalo pouze rozdílné prvky. Dále je nutné zajistit aby extrakce těchto prvků proběhla v lineárním čase a neovlivnila tak asymptotickou časovou složitost funkce Merge.

Extrakce se provádí ze seřazené části pole. Nejprve se přidá první prvek v poli. Poté se nalezne další prvek, který není roven žádnému z předchozích

extrahovaných prvků. To je v případě seřazeného pole triviální, protože pro tento prvek platí, že není roven předchozímu prvku v daném poli. Poté se provede rotace extrahovaného bloku s blokem mezi novým prvkem a extrahovaným blokem a znovu se hledá další prvek, dokud není extrahován daný počet prvků. Nakonec se provede rotace extrahovaného bloku s blokem nacházejícím se před extrahovaným blokem, tedy tak, aby se extrahovaný blok nacházel na začátku pole a jeho pozdější distribuce zpět do pole mohla být stabilní.

Každý z prvků pole se přesune maximálně jednou vlevo a jednou vpravo, zatímco prvky extrahovaného bloku se mohou přesunout vpravo až tolikrát, jaká je délka extrahovaného bloku a poté jednou vlevo. Pokud extrahujeme blok délky m z pole délky n, je tak výsledná asymptotická časová složitost operace $\mathcal{O}(m^2 + n)$, či-li pro zachování lineární složitosti vzhledem k délce pole n je maximální možná délka extrahovaného bloku $m = k\sqrt{n}$, kde k je konstanta, pro kterou platí $k \ll n$.

Extrakci lze provádět i z neseřazeného pole. Při kontrole, zda se prvek v extrahovaném bloku již nachází, se místo porovnání s předchozím prvkem provede binární vyhledávání v extrahovaném bloku. K tomu musí být extrahovaný blok seřazený a proto se každý nový prvek do bloku lineárně vkládá na správné místo. To nijak neovlivní asymptotický počet přesunů, ale počet porovnání vzroste na $\mathcal{O}(n \log m)$, což je pro $m = \sqrt{n}$ asymptoticky ekvivalentní s $\mathcal{O}(n \log n)$.

2.7.2 Rotation Merge

Tento algoritmus je uvedený v [10]. Algoritmus *Rotation Merge* je jednoduchý algoritmus na sloučení dvou seřazených bloků u a v. Pro první prvek u se pomocí binárního vyhledávání určí blok prvků z v, které jsou ostře menší než daný prvek. Tyto prvky z v se pak pomocí rotace s blokem u umístí bezprostředně před daný prvek, který je následně odebrán z bloku u, tj. nový blok u začíná prvkem z u následujícím po tomto prvku. Tak pokračujeme, dokud u není prázdné.

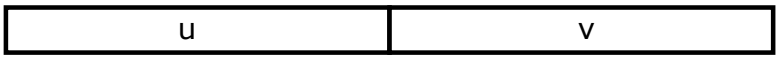
2.7.3 Block Rotation Merge

Tento algoritmus je uvedený v [10]. Algoritmus *Block Rotation Merge* sloučí seřazené bloky u a v následujícím způsobem. Rozdělí blok u na bloky $u_1u_2 \dots u_x$ zvolené délky d, kde u_1 je případný necelý blok. Blok v se poté rozdělí na bloky $v_1v_2 \dots v_x$ tak, aby pro prvky libovolného bloku v_i platilo, že jsou ostře menší než první prvek bloku u_{i+1}. Bloky jsou poté $x - 1$ rotacemi přeuspořádany do tvaru $u_1v_1u_2v_2 \dots u_xv_x$. Všechny odpovídající dvojice u_iv_i jsou pak sloučeny pomocí algoritmu *Rotation Merge*.

2.7.4 Block Merge

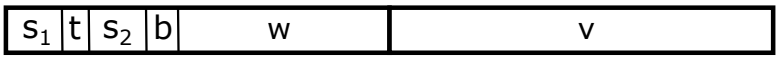
Následující odstavce jsou z převážné části převzaté z [10] a popisují průběh hlavní funkce Merge, která využívá předchozí algoritmy.

Označme levou část pole, které chceme sloučit, jako u a pravou část pole v, jako na obrázku 2.1. Zvolme délku bloků $d = \lfloor\sqrt{|u|}\rfloor$. Nejprve se u rozdělí na



Obrázek 2.1: Block Merge – 1. krok

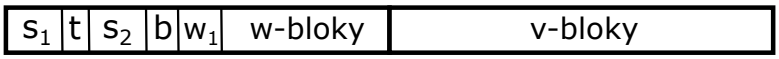
části $s_1 t s_2 u'$ tak, že s_1 a s_2 jsou bloky délky $\lfloor\frac{|u|}{d}\rfloor + \lfloor\frac{|v|}{d}\rfloor$ a t obsahuje všechny prvky následující bezprostředně po bloku s_1, které jsou rovny poslednímu prvku bloku s_1. Blok u' je pak zbytek z u bez $s_1 t s_2$. Poté z u' extrahujeme vnitřní pomocné pole b délky d a zbytek z u' po extrakci nazveme w, pole se tedy nachází ve stavu znázorněném na obrázku 2.2.



Obrázek 2.2: Block Merge – 2. krok

Následuje rozdělení bloků w a v na bloky délek d, nazveme je w-bloky a v-bloky. Případný necelý blok z w se nachází na začátku w a necelý blok z v se naopak nachází na konci bloku v. Aktuální stav pole lze vidět na obrázku 2.3. Bloky délky d jsou reorganizovány tak, aby platily následující podmínky:

- Vzájemné pořadí w-bloků zůstane nezměněné, stejně tak v-bloků.
- Pokud po v-bloku následuje w-blok, potom poslední prvek v-bloku je ostře menší než první prvek w-bloku
- Pokud po w-bloku následuje v-blok, potom první prvek w-bloku je menší nebo roven poslednímu prvku v-bloku.



Obrázek 2.3: Block Merge – 3. krok

To je uskutečněno postupnou záměnou bloků. Během této operace se mění vzájemné pořadí w-bloků. Pro stabilitu je tedy nutné zajistit správný výběr prvního w-bloku. To je uskutečněno zrcadlením pohybu w-bloků v pomocném poli b. To se totiž skládá z různých prvků a je seřazené, takže umístění w-bloku, který byl původně nejvíce vlevo, je možné zjistit z pozice aktuálního nejmenšího prvku v pomocném poli.

Samotná záměna probíhá následovně. Porovná se první prvek w-bloku, nalezeného způsobem popsaným v přechozím odstavci, s posledním prvek nejlevějšího v-bloku, který ještě nebyl umístěn. Pokud je mu menší nebo roven, prohodí se daný w-blok s aktuálně nejlevějším w-blokem. V opačném případě se prohodí daný v-blok s aktuálně nejlevějším w-blokem. Tak se pokračuje, dokud nebyly umístěny všechny w-bloky. Po této operaci se pole nachází ve stavu zobrazeném na obrázku 2.4.

Obrázek 2.4: Block Merge – 4. krok

V průběhu záměny bloků se do částí s_1 a s_2 zaznamenává, který blok pochází z w. Tato informace je uložena poměrně jednoduše. Díky existenci t platí, že všechny prvky z s_1 jsou menší než prvky z s_2. Záměnou x-tých prvků z s_1 a s_2 se pak indikuje, zda je x-tý blok w-blokem. To platí, pokud x-tý prvek z s_2 je menší než x-tý prvek z s_1.

Poté, co jsou bloky ve správném pořadí podle uvedených podmínek, začnou se postupně zprava procházet. Pokud se jedná o w-blok, navrátí se jemu odpovídající prvky v s_1 a s_2 do správného pořadí a provede se následující. Označme tento w-blok jako p a blok skládající se z prvků pocházejících z v následující bezprostředně po p jako q. Rozdělíme q na q_1q_2 tak, že pro prvky q_1 platí, že jsou ostře menší než první prvek p a prvky q_2 jsou mu větší nebo rovny. Provedeme rotaci bloku p s blokem q_1 a sloučíme p s q_2 pomocí vnitřního pomocného pole b.

Stav poté, co takto projdeme všechny w-bloky, je znázorněn na obrázku 2.5. V Levé části zbyde $s_1ts_2bw_1v'$, kde w_1 je necelý w-blok a v' je blok prvků

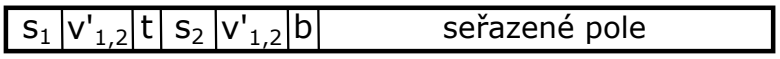
Obrázek 2.5: Block Merge – 5. krok

z v následující po tomto bloku. Rozdělíme v' na $v'_1v'_2$ tak, aby v'_1 obsahovalo všechny prvky z v', které jsou ostře menší než poslední prvek bloku s_2. Provedeme rotaci z $bw_1v'_1$ na v'_1bw_1. Dostaneme se tedy do situace z obrázku 2.6. Následně sloučíme w_1 s v'_2 s využitím pomocného pole b.

Obrázek 2.6: Block Merge – 6. krok

Obdobně blok v'_1 rozdělíme na $v'_{1,1}v'_{1,2}$ tak, aby $v'_{1,1}$ obsahovalo všechny prvky z v'_1, které jsou ostře menší než poslední prvek bloku s_1. Rotací změníme $s_1ts_2v'_{1,1}v'_{1,2}$ na $s_1v'_{1,1}ts_2v'_{1,2}$, poté se pole nachází ve stavu z obrázku

2.7. Následně sloučíme s_1 s $v'_{1,1}$ a s_2 s $v'_{1,2}$ algoritmem *Block Rotation Merge*



Obrázek 2.7: Block Merge – 7. krok

s blokem velikosti d. Nakonec algoritmem *Insertion sort* seřadíme pomocné pole b a pomocí algoritmu *Rotation Merge* ho sloučíme se zbytkem pole. Po těchto operacích je celé původní pole uv seřazené.

Pokud na začátku algoritmu je u' prázdné, rozdělíme v na v_1v_2 tak, aby v_1 obsahovalo všechny prvky z v, které jsou ostře menší než poslední prvek bloku s_1. Provedeme rotaci z $s_1ts_2v_1v_2$ na $s_1v_1ts_2v_2$. Pomocí algoritmu *Block-Rotation-Merge* sloučíme s_1 s v_1 a s_2 s v_2. Pokud je s_2 prázdné, sloučíme pouze první dvojici. Pokud se podařilo extrahovat pouze $d < \lfloor\sqrt{|u|}\rfloor$ prvků, změníme délku bloku na $\lfloor\frac{|u|}{d}\rfloor$ a jinak postupujeme stejně s tím rozdílem, že místo algoritmu na sloučení s pomocí vnitřního pomocného pole používáme algoritmus *Rotation Merge*.

Kapitola 3

Optimalizace

Tato kapitola obsahuje popis optimalizace algoritmů uvedených v kapitole 2. Dále obsahuje časy běhů algoritmů naměřené v průběhu tohoto procesu, protože některé z optimalizací a paralelizací závisí na úspěšnosti předchozích. Na úvod tato kapitola také obsahuje popis implementační a testovací platformy.

3.1 Implementační a testovací platforma

Nejprve tedy popíšeme softwarové a hardwarové aspekty pro měření výkonnosti algoritmů.

3.1.1 Programovací jazyk

Při volbě programovacího jazyka byl kladen důraz na nejvyšší potenciální efektivity výsledného programu. V úvahu tedy připadaly jazyky *C* a *C++*. Zvolen byl jazyk *C++*, protože oproti jazyku *C* obsahuje *template* funkce, které umožňují překladači místo volání funkcí na porovnání (prohození) dvou prvků vložit instrukce těchto funkcí přímo do kódu řadící funkce (*inline*). Programy byly kompilovány pomocí překladače *g++* verze 4.8.5 s následujícími přepínači:

- `-std=c++11`
- `-O3`
- `-fopenmp`
- `-march=core-avx-i`

3.1.2 OpenMP

Pro paralelizaci byla zvolena technologie *OpenMP*, která umožňuje snadnou paralelizaci programů napsaných v jazyce *C++* pomocí direktiv překladače.

Informace o OpenMP jsou čerpány z [12]. Mezi použité direktivy OpenMP v této práci patří:

- `#pragma omp parallel` – vytvoří tým vláken a zahájí paralelní blok
- `#pragma omp master` – určí, že následující blok se provede pouze v hlavním vlákně
- `#pragma omp task` – vytvoří novou úlohu pro zpracování jedním z vláken v týmu
- `#pragma omp taskwait` – způsobí čekání na dokončení podúloh aktuálně prováděné úlohy

Dále jsou v práci využity následující funkce:

- `omp_get_max_threads()` – vrátí maximální množství vláken vytvořených pro nový tým, který počet nestanoví explicitně
- `omp_set_num_threads()` – nastaví maximální množství vláken vytvořených pro nový tým, který počet nestanoví explicitně
- `omp_get_thread_num()` – vrátí pořadové číslo vlákna v aktuálním týmu

3.1.3 Hardwarové parametry

Všechna měření byla provedena na školním clusteru *star.fit.cvut.cz*. Konkrétně pak na konfiguraci 2x *Xeon E5-2620 v2* a 32 GB paměti RAM. Každý z těchto procesorů má 6 fyzických jader a podporuje technologii *Hyper-Threading*, která umožňuje běh až 2 vláken najednou na 1 fyzickém jádru. Dohromady může na této konfiguraci tedy běžet až 24 vláken najednou.

3.1.4 Vstupní data

Veškerá vstupní data jsou generována pomocí pseudonáhodného generátoru `std::mt19937_64` ze standardní knihovny *C++*. Generátor je vždy inicializován konstantní hodnotou 42 pro zajištění identických vstupů při každém měření. Vstupní data jsou těchto typů:

- `int`
- `long long`
- `char`
- `BigObject<4096>` – objekt velikosti 4096 B, který má výrazně delší dobu potřebnou pro prohození dvou prvků, než pro jejich porovnání.

- `SlowCompare` – objekt, který má výrazně delší dobu potřebnou pro porovnání dvou prvků, než pro jejich prohození.

- `BigSlow<4096>` – objekt velikosti 4096 B, který má výrazně delší dobu potřebnou pro prohození i porovnání dvou prvků oproti elementárním prvkům jako je `int`.

3.1.5 Způsob zpracování naměřených výsledků

Pro účely přehledného zpracování naměřených výsledků ve formě tabulky provedeme následující označení vstupních dat:

- Data 1 – pole typu `int` o délce 10 000 000

- Data 2 – pole typu `int` o délce 100 000 000

- Data 3 – pole typu `long long` o délce 7 000 000

- Data 4 – pole typu `char` o délce 20 000 000

- Data 5 – pole objektů `BigObject<4096>` o délce 100 000

- Data 6 – pole objektů `SlowCompare` o délce 400 000

- Data 7 – pole objektů `BigSlow<4096>` o délce 100 000

Dále je prováděno měření času pro náhodně uspořádané, již seřazené a opačně seřazené pole. Tyto různé typy vstupních posloupností označíme jako Typ 1, Typ 2 a Typ 3 resp.

Všechny naměřené hodnoty v tabulkách jsou uvedeny v sekundách. Hodnoty, které byly alespoň o 5 % rychlejší, než hodnoty porovnávaného algoritmu, jsou vyznačeny zelenou barvou. Naopak hodnoty, které byly alespoň o 5 % pomalejší, jsou vyznačeny červenou barvu. Ostatní hodnoty nejsou nijak zvýrazněny.

Pokud jsou porovnávány časy dosažené stejným algoritmem pro různé hodnoty nějakého parametru, je tak provedeno formou grafu. Hodnoty, které jsou na grafu znázorněny, jsou časy dosažené při řazení vstupu Typ 1, Data 2, nebo z nich, v případě měření zrychlení paralelního algoritmu, vycházejí.

3.2 Out-of-place Mergesort

Nyní popíšeme optimalizaci algoritmů *out-of-place Mergesortu*.

3.2.1 Základní algoritmus

Základní algoritmus pro Mergesort se od uvedených algoritmů 2.2 a 2.3 příliš neliší. Hlavním rozdílem je alokace pomocného pole před samotným voláním funkce Mergesort, aby se předešlo opakované alokaci ve funkci Merge. Časy dosažené tímto algoritmem jsou zaznamenány v tabulce 3.1.

Tabulka 3.1: Out-of-place Mergesort – základní verze

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,141	12,861	0,849	2,289	1,846	1,004	2,118
Typ 2	0,671	7,693	0,539	1,250	1,864	0,754	2,102
Typ 3	0,686	7,860	0,551	1,370	1,827	0,695	2,056

3.2.1.1 Využití pomocného pole

První zřejmou optimalizací je nepřesouvat výsledek z pomocného pole zpět do pole původního, ale ponechat ho v pomocném poli a v dalším volání funkce Merge přesouvat prvky z pomocného pole do pole původního. To je možné například záměnou ukazatele na původní pole s ukazatelem na pole pomocné při rekurzivním volání. Informace, zda jsou ukazatele zaměněné, je pak možné uložit v novém parametru funkce Mergesort. Je třeba zajistit, aby při posledním volání funkce Merge, tj. Merge, jehož výsledkem je seřazené celé pole, bylo cílové pole polem původním.

Časy dosažené po této optimalizaci jsou zaznamenány v tabulce 3.2. Ke zrychlení došlo ve všech případech. K největšímu zrychlení pak došlo u největších datových typů, kde je samozřejmě doba přesunů, které byly touto optimalizací odstraněny, nejdelší.

Tabulka 3.2: Out-of-place Mergesort – zaměňování cílového a pomocného pole

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,141	12,861	0,849	2,289	1,846	1,004	2,118
Typ 2	0,671	7,693	0,539	1,250	1,864	0,754	2,102
Typ 3	0,686	7,860	0,551	1,370	1,827	0,695	2,056
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,044	11,665	0,736	2,146	1,011	0,952	1,250
Typ 2	0,584	6,599	0,435	1,136	1,027	0,649	1,216
Typ 3	0,599	6,753	0,448	1,259	0,994	0,592	1,184

3.2.1.2 Zjednodušení cyklu ve funkci Merge

V uvedeném algoritmu 2.3 se kontrola, zda už nebyla vyčerpána jedna z polovin, provádí společně pro obě dvě. To je samozřejmě zbytečné, protože ji lze

provádět pouze pro tu část, z které byl právě odebrán prvek.

Časy dosažené po této změně jsou zaznamenány v tabulce 3.3. Pro většinu náhodných vstupů došlo k výraznému zpomalení. Naopak ke zrychlení došlo pro seřazené a obráceně seřazené vstupy a také pro řazení náhodného pole typu `char`. Tyto skupiny mají společné to, že při slučování se vždy do výstupního pole přesune mnoho prvků z jedné z částí, poté mnoho prvků z druhé části atd. Zrychlení je tedy pravděpodobně podmíněno úspěšnou predikcí skoku. Vzhledem k výraznému zpomalení v ostatních případech se však tato optimalizace nevyplatila a nebude uplatněna dále.

Tabulka 3.3: Out-of-place Mergesort – zjednodušení cyklu ve funkci Merge

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,044	11,665	0,736	2,146	1,011	0,952	1,250
Typ 2	0,584	6,599	0,435	1,136	1,027	0,649	1,216
Typ 3	0,599	6,753	0,448	1,259	0,994	0,592	1,184
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,439	16,227	0,903	1,830	1,011	0,916	1,251
Typ 2	0,259	2,821	0,214	0,471	1,026	0,646	1,217
Typ 3	0,298	3,250	0,245	0,491	0,993	0,591	1,187

3.2.1.3 Zakončení rekurze Insertion sortem

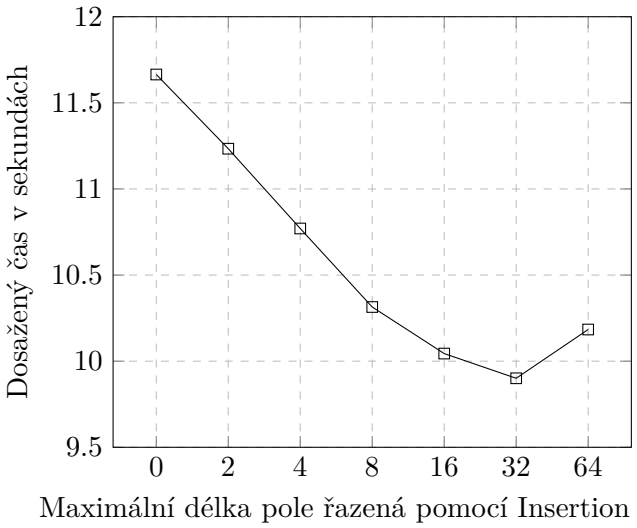
Další optimalizací je pro určité délky pole nepoužívat Mergesort, ale jiný algoritmus, jako je např. *Insertion sort*. Jelikož je Insertion sort také stabilní algoritmus, Mergesort tím stabilitu neztratí. Asymptotická časová složitost Insertion sortu je oproti Mergesortu horší, pro krátká pole má však lepší vlastnosti a je pro ně tedy rychlejší. Je totiž mnohem jednodušší a má tedy výrazně nižší režii.

Pro určení délky pole, která se budou řadit pomocí Insertion sortu, byla provedena měření doby řazení objektů typu `int`. Jako hranice byly voleny mocniny dvou z důvodu povahy Mergesortu, tedy protože délka řazeného pole v každé následující úrovni rekurze je poloviční. Dosažené časy lze vidět na obrázku 3.1.

Jako nejlepší se pak jeví hranice délky 32. Tabulka 3.4 zachycuje časy dosažené při použití konstanty 32. Ke zrychlení došlo ve většině případů. Výjimku tvoří pole objektů typu `SlowCompare`, kde se projevil výrazně větší počet porovnání. Větší počet přesunů pak nebyl velkým problémem, díky efektivní práci s cache pamětí.

3.2.1.4 Rozbalení cyklu ve funkci Merge

První cyklus ve funkci Merge je netradičního tvaru a kompilátor ho nedokáže sám automaticky rozbalit, musí se tedy rozbalit explicitně. Toho lze docílit



Obrázek 3.1: Out-of-place Mergesort – doba řazení při použití Insertion sortu pro různé délky pole

Tabulka 3.4: Out-of-place Mergesort – doba řazení při použití Insertion sortu pro pole délky 32 a méně

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,044	11,665	0,736	2,146	1,011	0,952	1,250
Typ 2	0,584	6,599	0,435	1,136	1,027	0,649	1,216
Typ 3	0,599	6,753	0,448	1,259	0,994	0,592	1,184
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,871	9,905	0,608	1,767	1,001	0,989	1,247
Typ 2	0,423	4,851	0,312	0,820	0,892	0,622	1,085
Typ 3	0,512	6,085	0,409	0,940	1,093	0,779	1,326

např. vytvořením zarážek pro řídící proměnné a zopakováním těla cyklu. Zarážky jsou původní meze snížené o počet zopakování, kvůli zabránění přístupu mimo pole v nejhorším případě, tedy když se bude v každém kroku přesouvat ze stejné části pole. Po tomto upraveném cyklu je ponechán původní cyklus, který dokončí zbytek po zarážkách.

V tabulce 3.5 lze vidět časy naměřené při použití této optimalizace. Došlo k mírnému zpomalení ve většině případů. Důvodem jsou nejspíše výpadky instrukční paměti z důvodu výrazně delšího kódu. Výjimku tvoří řazení pole typu `SlowCompare`. Důvodem může být složitější funkce na porovnání prvků, která mohla způsobovat výpadky už před rozbalením cyklu.

Tabulka 3.5: Out-of-place Mergesort – rozbalení Merge cyklu

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,871	9,905	0,608	1,767	1,001	0,989	1,247
Typ 2	0,423	4,851	0,312	0,820	0,892	0,622	1,085
Typ 3	0,512	6,085	0,409	0,940	1,093	0,779	1,326
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,889	10,109	0,615	1,807	1,004	0,974	1,247
Typ 2	0,427	4,931	0,313	0,832	0,892	0,638	1,085
Typ 3	0,519	6,171	0,413	0,956	1,094	0,796	1,324

3.2.1.5 Merge od konce pole

Tato optimalizace zlepšuje využití datové cache paměti. Pokud uvažujeme slučování dlouhých polí, které se nevejdou celé do cache paměti, zůstane po proběhnutí funkce Merge v cache paměti konec první a druhé poloviny pole, z kterého přesouváme, a konec pole, do kterého přesouváme. Toho lze využít v následujícím volání funkce Merge pomocí slučování od konce pole. Vyhneme se tím některým načítáním z hlavní paměti nebo z cache vyšší úrovně. Zaručeně nám totiž v cache paměti zůstane konec pravé poloviny aktuálního zdrojového pole a ve většině případů i konec výstupního pole.

Slučování od konce pole probíhá obdobně jako klasická varianta. Porovnají se poslední prvky levé a pravé poloviny pole. Větší z nich se přesune na konec výstupního pole. Pokud jsou si prvky rovny, je pro zachování stability nutné přesunout prvek z pravé poloviny, na rozdíl od klasické varianty, kdy se v případě rovnosti přesouvá prvek z levé poloviny. To se provádí, dokud se nevyčerpají všechny prvky jedné z polovin. Zbytek druhé poloviny se pak stejně jako v klasické variantě pouze přesune do výstupního pole.

Po slučování od konce pole naopak zbyde v cache paměti začátek první a druhé poloviny zdrojového pole a začátek cílového pole. V tomto případě tedy nejlépe využije cache paměť naopak klasická varianta funkce Merge. Ideální je nejprve seřadit druhou polovinu a poté tu první, aby nám tak v cache paměti ve většine případů zůstal začátek nového cílového pole a s ním zaručeně začátek levé poloviny nového zdrojového pole.

Optimálního využití cache paměti lze tedy docílit střídáním klasické varianty funkce Merge s variantou Merge od konce pole. Toto střídání však může zapříčinit výpadky instrukční cache paměti, protože používaný kód bude přibližně dvakrát delší, než před touto optimalizací. Je tedy vhodné tyto varianty střídat jen v nejvyšších úrovních, kdy jsou případné načítání do instrukční paměti zanedbatelné vzhledem k ušetřeným načítáním datové cache paměti. V nižších úrovních rekurze se zpravidla obě pole vejdou do cache paměti a střídání obou variant tak nemá smysl, proto je zde vhodnější používat pouze jednu variantu.

Časy naměřené po aplikaci této optimalizace jsou zaznamenány v tabulce 3.6. Ve většině případů došlo k mírnému zpomalení. Důvod není zřejmý. Výjimku pak tvořily speciální objekty, které pracují s výrazně větší pamětí na jeden prvek. Koncept optimalizace je tedy pravděpodobně správný. Za zmínku také stojí, že v průběhu psaní této práce došlo ke změně verze překladače na testovacím serveru Star a na předchozí verzi tato optimalizace přinesla zrychlení.

Tabulka 3.6: Out-of-place Mergesort – Merge od konce pole

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,871	9,905	0,608	1,767	1,001	0,989	1,247
Typ 2	0,423	4,851	0,312	0,820	0,892	0,622	1,085
Typ 3	0,512	6,085	0,409	0,940	1,093	0,779	1,326
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,872	9,916	0,609	1,767	0,996	0,941	1,243
Typ 2	0,430	4,927	0,315	0,851	0,882	0,608	1,083
Typ 3	0,521	6,137	0,410	0,920	1,088	0,799	1,329

3.2.2 Mergesort s poloviční pomocným polem

Tato varianta bude optimalizována některými principy, které byly využity při optimalizaci základního algoritmu. Dále pak budou navrženy právě různé způsoby pro eliminaci přesunu jedné z polovin před slučováním. V tabulce 3.7 jsou zaznamenány časy dosažené při použití neoptimalizované varianty. Algoritmus je rychlejší oproti neoptimalizované klasické verzi ve všech případech. V porovnání s verzí po optimalizaci ze sekce 3.2.1.1 je v některých případech mírně rychlejší ale v ostatních případech o něco málo pomalejší.

Tabulka 3.7: Out-of-place Mergesort – Poloviční pomocné pole

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,033	11,660	0,745	2,051	1,249	0,976	1,501
Typ 2	0,549	6,256	0,407	1,028	0,855	0,629	1,041
Typ 3	0,612	7,037	0,469	1,163	1,279	0,647	1,485

3.2.2.1 Využití Insertion sortu pro krátká pole

Stejně jako v klasické variantě lze pro krátká pole využít Insertion sortu a zakončit s ním rekurzi. Důvodem k této optimalizaci je opět výrazně nižší režie. Z důvodu podobného dosaženého času nemá smysl znovu provádět testování pro volbu délky pole řazeného pomocí Insertion sortu a lze rovnou použít konstantu 32.

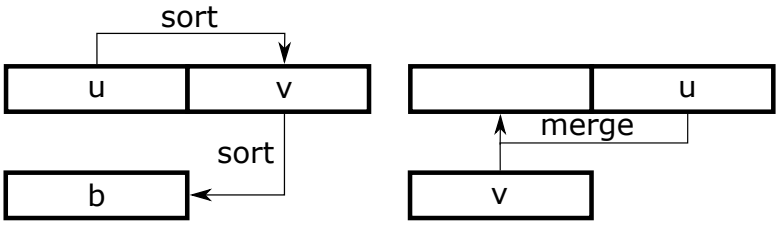
V tabulce 3.8 lze vidět časy dosažené po této optimalizaci. V porovnání s klasickou verzí po aplikaci této optimalizace je o trochu pomalejší. Je však důležité mít na paměti, že před každým slučováním se stále přesouvá levá polovina pole do jiného umístění.

Tabulka 3.8: Out-of-place Mergesort – Použití Insertion sortu

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,033	11,660	0,745	2,051	1,249	0,976	1,501
Typ 2	0,549	6,256	0,407	1,028	0,855	0,629	1,041
Typ 3	0,612	7,037	0,469	1,163	1,279	0,647	1,485
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,876	10,097	0,640	1,801	1,247	0,977	1,516
Typ 2	0,410	4,779	0,298	0,823	0,769	0,607	0,966
Typ 3	0,498	6,062	0,433	0,959	1,365	0,836	1,624

3.2.2.2 Eliminace zbytečných přesunů

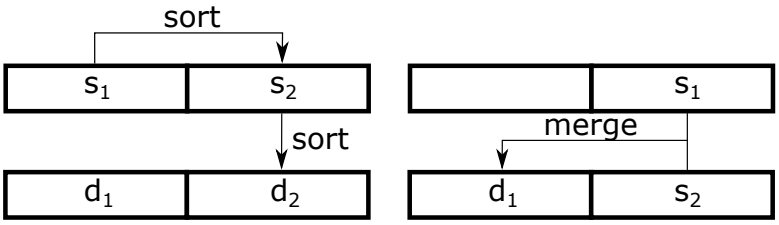
Zjevnou nevýhodou této varianty oproti klasické je právě nutnost přesunu první poloviny pole do pomocného pole před slučováním. Tomuto přesunu se dá však vyhnout např. následujícím způsobem. Rozdělíme pole na u a v, kde $|u| = \lfloor \frac{n}{2} \rfloor$ a $|v| = \lceil \frac{n}{2} \rceil$. Poté seřadíme v do pomocného pole a u seřadíme do v tak, jak je zobrazeno na obrázku 3.2. Je tedy potřebné pomocné pole velikosti $b = \lceil \frac{n}{2} \rceil$ namísto $\lfloor \frac{n}{2} \rfloor$. Pokud bychom za účelem zachování velikosti pomocného pole $b = \lfloor \frac{n}{2} \rfloor$ zvolili opačné dělení, mohlo by se zdrojové a cílové pole pro řazení u překrývat jedním prvkem.



Obrázek 3.2: Postup pro řazení vstupního pole

Zmíněná řazení provádíme obdobně. Zdrojové pole s rozdělíme na s_1 a s_2, kde $|s_1| = \lfloor \frac{|s|}{2} \rfloor$ a $|s_2| = \lceil \frac{|s|}{2} \rceil$. Cílové pole d rozdělíme na d_1 a d_2, kde $|d_1| = \lfloor \frac{|d|}{2} \rfloor$ a $|d_2| = \lceil \frac{|d|}{2} \rceil$. Poté tímto algoritmem rekurzivně seřadíme s_2 do d_2 a následně s_1 do s_2. Ty pak sloučíme do d_1. Tento postup je znázorněn na obrázku 3.3.

Zakončení rekurze pak provedeme tak, že se zastavíme pro dvojnásobnou délku pole potřebnou pro Insertion sort. Insertion sortem seřadíme s_1 a s_2 a



Obrázek 3.3: Řazení ze zdrojového pole do pole cílového

sloučíme je do d. Jak je vidět v tabulce 3.9 tato optimalizace přinesla zrychlení a předčila nejlepší sekvenční verzi základního algoritmu ve všech případech s výjimkou opačně seřazených objektů typu `SlowCompare`. Důvod této výjimky není zřejmý.

Tabulka 3.9: Out-of-place Mergesort – Eliminace zbytečných přesunů

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,876	10,097	0,640	1,801	1,247	0,977	1,516
Typ 2	0,410	4,779	0,298	0,823	0,769	0,607	0,966
Typ 3	0,498	6,062	0,433	0,959	1,365	0,836	1,624
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,698	0,596	1,765	0,921	0,949	1,156
Typ 2	0,387	4,418	0,264	0,799	0,501	0,509	0,651
Typ 3	0,464	5,623	0,376	0,931	0,996	0,876	1,234

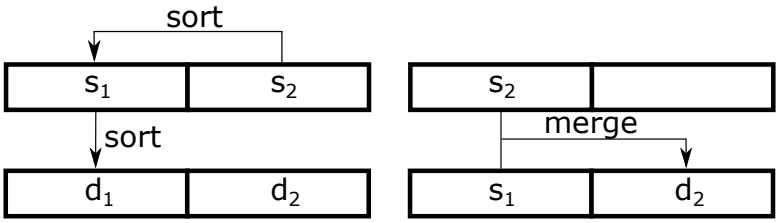
3.2.2.3 Zlepšení využití cache paměti

Obdobně jako v klasické variantě lze i tuto variantu optimalizovat tím, že budeme střídat klasické slučování a slučování od konce. Pokud však budeme používat slučování od konce, bude postup trochu odlišný. Rozdělení zůstane obdobné s tím rozdílem, že případné delší části budou ty levé. Tedy zdrojové pole s rozdělíme na s_1 a s_2, kde $|s_1| = \lceil \frac{|s|}{2} \rceil$ a $|s_2| = \lfloor \frac{|s|}{2} \rfloor$. Cílové pole d rozdělíme na d_1 a d_2, kde $|d_1| = \lceil \frac{|d|}{2} \rceil$ a $|d_2| = \lfloor \frac{|d|}{2} \rfloor$. Následně s_1 seřadíme do d_1, s_2 do s_1 a seřazené části sloučíme do d_2. Toto je ukázáno na obrázku 3.4.

Časy dosažené po této optimalizaci jsou zaznamenány v tabulce 3.10. V některých případech došlo k mírnému zrychlení, v jiných naopak k mírnému zpomalení. Celkově se pak tato optimalizace spíše nevyplatí než vyplatí.

3.2.2.4 Další zlepšení využití cache paměti

Pro rozhodnutí, kdy používat klasické slučování a kdy slučovat od konce, budeme uvažovat, že velikost paměti cache $c < \lceil \frac{|s|}{2} \rceil$. Po provedení operace merge



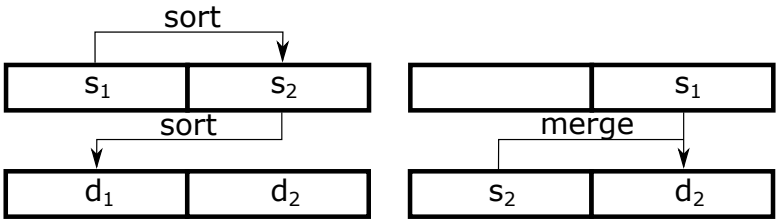
Obrázek 3.4: Postup při slučování od konce pole

Tabulka 3.10: Out-of-place Mergesort – Zlepšení využití cache paměti

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,698	0,596	1,765	0,921	0,949	1,156
Typ 2	0,387	4,418	0,264	0,799	0,501	0,509	0,651
Typ 3	0,464	5,623	0,376	0,931	0,996	0,876	1,234
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,856	9,802	0,596	1,752	0,921	0,942	1,161
Typ 2	0,384	4,427	0,264	0,790	0,503	0,515	0,652
Typ 3	0,500	6,062	0,376	0,895	0,994	0,837	1,221

(backmerge) tedy v cache paměti zůstane konec (začátek) cílového pole a konec (začátek) původního zdrojového pole. Pro sort, který je v pořadí druhý, tedy vždy chceme slučovat opačným způsobem, než jaký budeme používat v aktuální úrovni rekurze.

Pro optimální volbu pro první sort je zapotřebí vědět, s jakou částí pole bude druhý sort pracovat jako první. K tomuto účelu trochu pozměníme délky částí a cílová pole jednotlivých řazení pokud je budeme slučovat od konce. Tedy zdrojové pole s rozdělíme na s_1 a s_2, kde $|s_1| = \lfloor \frac{|s|}{2} \rfloor$ a $|s_2| = \lceil \frac{|s|}{2} \rceil$. Cílové pole d rozdělíme na d_1 a d_2, kde $|d_1| = \lceil \frac{|d|}{2} \rceil$ a $|d_2| = \lfloor \frac{|d|}{2} \rfloor$. To protože s_2 seřadíme do d_1 a s_1 do s_2. Následovně s_1 a s_2 sloučíme do d_2 tak, jak je znázorněno na obrázku 3.5.



Obrázek 3.5: Alternativní postup při slučování od konce pole

Touto změnou přijdeme o ušetření přesunů v neobvyklém případě, kdy s_1 obsahuje na začátku několik identických prvků a s_2 tímto prvkem začíná také.

Tato změna nám však především umožní určit, s jakou částí pole bude každý sort pracovat jako první, konkrétně každý sort tak bude nejprve pracovat s koncem pole. Proto pro první sort budeme vždy používat klasickou variantu slučování, která ponechá v cache paměti konec pole, který bude využit jako první při následujícím řazení.

Tabulka 3.11 obsahuje časy naměřené po aplikaci této optimalizace. Ve většině případů došlo k mírnému zrychlení oproti předchozí optimalizaci, ale ke zpomalení vzhledem k algoritmu ze sekce 3.2.2.2. Tato optimalizace tedy nebyla úspěšná.

Tabulka 3.11: Out-of-place Mergesort – Další zlepšení využití cache paměti

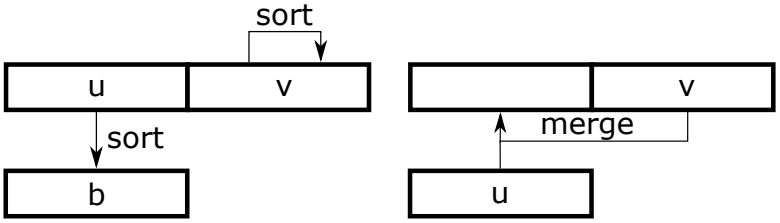
Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,698	0,596	1,765	0,921	0,949	1,156
Typ 2	0,387	4,418	0,264	0,799	0,501	0,509	0,651
Typ 3	0,464	5,623	0,376	0,931	0,996	0,876	1,234
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,855	9,794	0,596	1,751	0,920	0,938	1,159
Typ 2	0,396	4,586	0,273	0,814	0,531	0,521	0,686
Typ 3	0,484	5,858	0,364	0,885	0,964	0,824	1,184

3.2.2.5 Dvě varianty Mergesortu

Lze si všimnout, že v optimalizaci navržené v sekci 3.2.2.2 jsou použity dvě varianty funkce Mergesort. První v nejvyšší vrstvě rekurze, která opravdu využívá pouze poloviční velikost pomocného pole. Druhá varianta, která je použita v ostatních případech, však využívá pomocné pole (označeno jako cílové pole) stejné délky jako zdrojové pole. Vhodnou úpravou algoritmu lze využívat první variantu i v nižších úrovních rekurze. Důvodem, proč tak nebylo učiněno hned v prvním případě, je právě existence dvou způsobů řazení, tedy dvou funkcí, a jejich pravidelné střídání, což vede přibližně k zdvojnásobení délky kódu a nemusí tedy nutně být rychlejší.

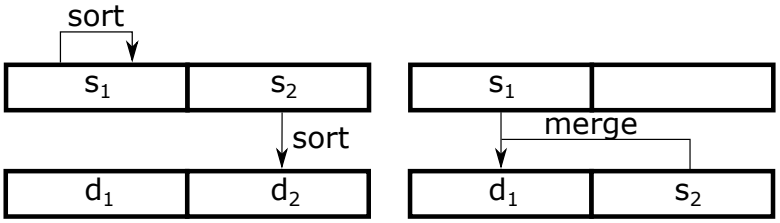
Zmíněná úprava algoritmu vypadá následovně. Rozdělíme pole na u a v, kde $|u| = \lfloor \frac{n}{2} \rfloor$ a $|v| = \lceil \frac{n}{2} \rceil$. Poté pomocí druhé varianty seřadíme u do pomocného pole b (tentokrát tak stačí $|b| = \lfloor \frac{n}{2} \rfloor$), v však seřadíme tak, aby zůstalo na svém místě, tedy první variantou. Jako pomocné pole lze využít původní umístění u, nebo obrátit pořadí operací a využít b. Nakonec samozřejmě sloučíme u a v do jejich původního umístění. Toto je znázorněno na obrázku 3.6.

Upravit lze pak i druhou variantu tak, aby využívala pro řazení první variantu. Zdrojové pole s rozdělíme na s_1 a s_2, kde $|s_1| = \lfloor \frac{|s|}{2} \rfloor$ a $|s_2| = \lceil \frac{|s|}{2} \rceil$. Cílové pole d rozdělíme na d_1 a d_2, kde $|d_1| = \lfloor \frac{|d|}{2} \rfloor$ a $|d_2| = \lceil \frac{|d|}{2} \rceil$. Poté s_2 seřadíme do d_2 a s_1 seřadíme pomocí první varianty, kde jako pomocné pole



Obrázek 3.6: Postup první varianty funkce Mergesort

můžeme při vhodném pořadí operací použít jakoukoliv z částí. Postup druhé varianty lze vidět na obrázku 3.7.



Obrázek 3.7: Postup druhé varianty funkce Mergesort

Problémem je tedy optimální pořadí operací a zvolení pomocných polí s ohledem na nejlepší využití cache paměti. Nejprve se zaměříme na druhou variantu. V té je totiž jednoznačné, že nejdříve seřadíme s_2 do d_2, neboť s původním umístěním s_2 se při následujícím slučování nijak nepracuje. Jako pomocné pole pro následující řazení pak můžeme zvolit původní umístění s_2, které je z části v cache paměti z předchozího řazení, nebo d_1, které bude využito při následujícím slučování. Jelikož pro řazení s_1 je volena varianta, která využívá menší množství paměti, zvolíme jako pomocné pole d_1, protože ve většině případů tak v cache paměti zůstane více pro následující slučování, než zůstalo pro řazení s_1 po seřazení s_2.

Pro první variantu pak nejprve seřadíme v a poté u. Prvním důvodem je zachování pořadí, kde ve druhé variantě se též začíná druhou polovinou pole. Druhým důvodem je pak, že v se řadí pomocí první varianty, tedy ve většině případů zůstane z b, které je následně využito pro řazení u, více v cache paměti, než při opačném postupu.

V tabulce 3.12 jsou zachyceny časy naměřené při použití této optimalizace. Tato varianta přinesla zrychlení vzhledem k původní variantě, bylo však menší než zrychlení varianty ze sekce 3.2.2.2.

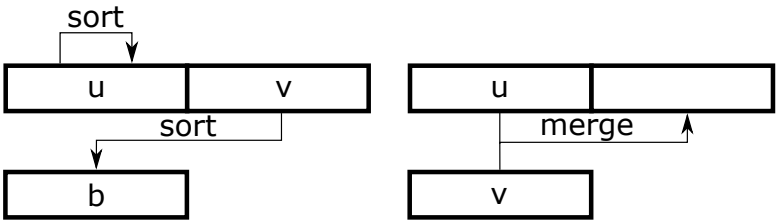
3.2.2.6 Aplikace slučování od konce v předchozí variantě

I v této variantě se střídáním slučování od konce a klasického slučování zlepší využití cache paměti. Je třeba vytvořit varianty pro obě verze *Mergesortu* uve-

Tabulka 3.12: Out-of-place Mergesort – Dvě varianty Mergesortu

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,698	0,596	1,765	0,921	0,949	1,156
Typ 2	0,387	4,418	0,264	0,799	0,501	0,509	0,651
Typ 3	0,464	5,623	0,376	0,931	0,996	0,876	1,234
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,861	9,805	0,602	1,770	0,896	0,943	1,134
Typ 2	0,387	4,434	0,261	0,803	0,439	0,504	0,585
Typ 3	0,483	5,786	0,374	0,935	1,008	0,859	1,233

dené v předchozí sekci. Začněme s první variantou. Abychom mohli slučovat od konce, je třeba mimo původní pole přesunout pravou polovinu. Proto změníme délky bloků na $u = \lceil \frac{n}{2} \rceil$ a $v = \lfloor \frac{n}{2} \rfloor$. Poté v seřadíme do b a u seřadíme do původního umístění, tak jak je vyobrazeno na obrázku 3.8.



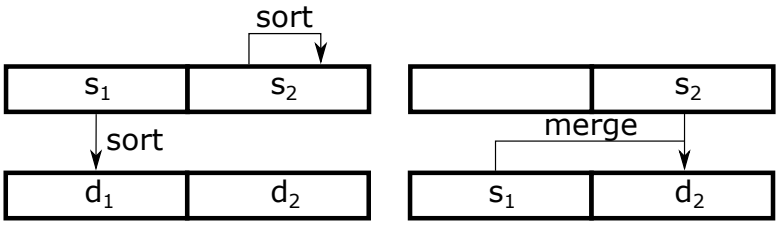
Obrázek 3.8: První varianta funkce Mergesort – slučování od konce

Dále se zaměříme na pořadí jednotlivých řazení. Jelikož ve druhé variantě první a druhé řazení nevyužívá stejné části pole, není pro nás predikce části pole, která bude používána při následujícím řazení, tak důležitá. Nemusíme tedy začínat s pravou polovinou pole. Naopak začneme s levou polovinou, protože je řazena první variantou a využívá tak méně paměti, či-li zanechá více prvků pro následující řazení, které bude využívat stejný prostor pro pomocné pole (cílové pole).

Obdobně pozměníme i druhou variantu. Vzhledem ke stejné délce zdrojového i cílového pole není potřeba měnit délky bloků. Pouze tentokrát seřadíme blok s_1 do d_1 a s_2 do svého původního umístění a následně sloučíme do cílového pole. Toto je znázorněno na obrázku 3.9.

Poslední otázkou pak zůstává pořadí jednotlivých řazení. Ze stejného důvodu jako u varianty s normálním slučováním začneme s řazením s_1 do d_1, protože právě původní oblast s_1 nebude využita při následujícím slučování. Poté s_2 seřadíme za využití d_2 jako pomocného pole, protože d_2 bude cílovým polem následujícího slučování.

V tabulce 3.13 jsou zaznamenány časy po této optimalizaci. Ve většině případů přinesla zrychlení vzhledem k verzi bez slučování od konce. Opět



Obrázek 3.9: Druhá varianta funkce Mergesort – slučování od konce

však bylo menší v porovnání s verzí ze sekce 3.2.2.2.

Tabulka 3.13: Out-of-place Mergesort – Dvě varianty Mergesortu se slučováním od konce

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,698	0,596	1,765	0,921	0,949	1,156
Typ 2	0,387	4,418	0,264	0,799	0,501	0,509	0,651
Typ 3	0,464	5,623	0,376	0,931	0,996	0,876	1,234
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,858	9,768	0,598	1,766	0,893	0,934	1,131
Typ 2	0,387	4,428	0,262	0,799	0,437	0,511	0,582
Typ 3	0,476	5,746	0,369	0,911	1,004	0,858	1,229

3.3 In-place Mergesort

Nyní popíšeme optimalizaci algoritmů in-place Mergesortu a algoritmů, které jsou těmito algoritmy využívány.

3.3.1 RecMerge

Tento algoritmus byl uveden ze dvou důvodů. Jedním z nich je velké množství rotací, které tento algoritmus provádí, je tedy vhodné ho použít při optimalizaci algoritmu pro rotaci bloků. Druhým důvodem je, že poskytuje základ pro paralelizaci nadcházejících algoritmů. Optimalizace sekvenční verze tohoto algoritmu tak není velmi důležitá a nebude tedy detailně popsána, místo ní se tato sekce bude zabývat právě optimalizací rotace bloků. Nejprve v tabulce 3.14 uvedeme časy dosažené při použití zmíněného algoritmu 2.4 používajícího převrácení pole.

Dále popíšeme alternativní algoritmus z [13], který postupuje následujícím způsobem. Delší z bloků uv rozdělíme na dvě části tak, aby jedna z částí byla stejně dlouhá jako kratší blok a pojmenujeme ji u', resp. v'. Pořadí bloků pak bude $u'uv$ resp. uvv'. Zaměníme kratší z bloků s novým blokem, tedy vznikne

Tabulka 3.14: In-place Mergesort – Rotace bloků pomocí převrácení

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	6,009	69,397	4,108	6,837	3,651	1,452	4,263
Typ 2	0,178	1,887	0,135	0,358	0,005	0,164	0,045
Typ 3	1,795	22,067	1,317	2,281	6,144	1,496	7,146

vuu' resp. $v'vu$. V tento moment je kratší z původních bloků na svém finálním místě. Zbylé bloky přejmenujeme opět na uv a pokračujeme stejným způsobem do té doby, než $|u| = |v|$. V ten moment pouze zaměníme u s v a rotace je dokončena.

Tento algoritmus funguje, protože kratší z bloků je vždy umístěn na své finální místo, přičemž po této operaci vzniknou z druhého bloku dva fragmenty, které tvoří identický problém tomu předchozímu. Nakonec zbydou dva bloky stejné délky, které jsou pouze zaměněny, jelikož rotace bloků identické délky je ekvivalentní s jejich záměnou.

Výhoda tohoto algoritmu spočívá v lepším využití cache paměti, kdy v případě kde jeden z bloků je vícenásobně větší než ten druhý, se při záměně bloků opakovaně využívá stejná část paměti, tedy původní oblast u' resp. v'. Naopak nevýhodou tohoto algoritmu je eventuelní degradace na záměnu velmi krátkých bloků, v nejhorším případě bloků s jedním prvkem. Pokud se podíváme na tabulku 3.15, kde jsou zachyceny časy při použití tohoto algoritmu, zjistíme, že tento algoritmus si vedl o něco hůře než předchozí.

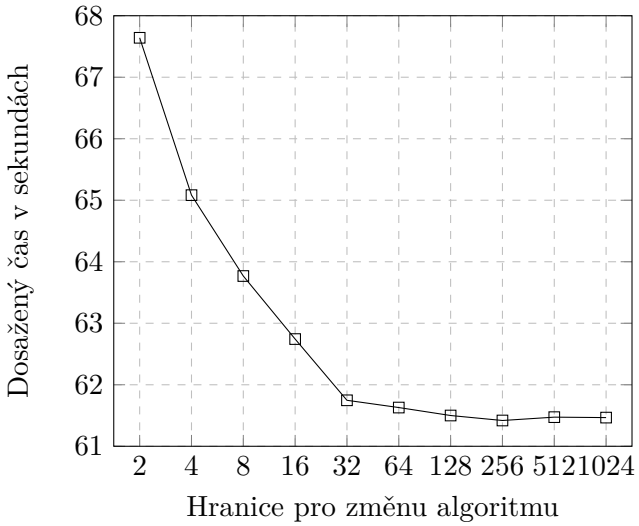
Tabulka 3.15: In-place Mergesort – Rotace bloků pomocí zaměňování

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	6,009	69,397	4,108	6,837	3,651	1,452	4,263
Typ 2	0,178	1,887	0,135	0,358	0,005	0,164	0,045
Typ 3	1,795	22,067	1,317	2,281	6,144	1,496	7,146
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	6,193	71,724	4,332	7,407	3,332	1,384	3,788
Typ 2	0,199	2,138	0,143	0,372	0,005	0,168	0,045
Typ 3	2,028	24,128	1,297	2,355	4,942	1,063	5,396

3.3.1.1 Hybridní přístup

Jelikož algoritmus využívající záměnu bloků v každém kroku řeší stejný problém, tj. rotaci bloků, lze ho v určitém místě nahradit jiným algoritmem, tedy např. algoritmem využívajícím převrácení. Zkusíme tedy v momentě, kdy délka kratšího bloku klesne pod určitou hranici, přepnout na tento algoritmus. Na obrázku 3.10 lze vidět časy dosažené při daných hranicích. Jako nejlepší

se jeví hranice 256, pro kterou byly zaznamenány podrobnější výsledky do tabulky 3.16.



Obrázek 3.10: In-place Mergesort – doba řazení při hybridním přístupu pro danou hranici

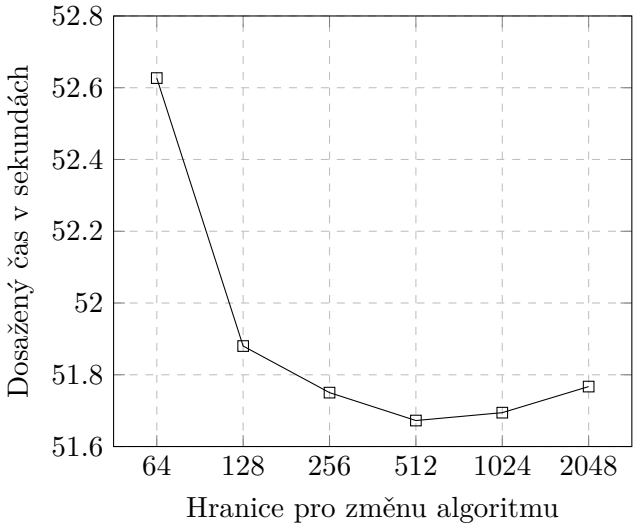
Tabulka 3.16: In-place Mergesort – Rotace bloků hybridním přístupem

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	6,009	69,397	4,108	6,837	3,651	1,452	4,263
Typ 2	0,178	1,887	0,135	0,358	0,005	0,164	0,045
Typ 3	1,795	22,067	1,317	2,281	6,144	1,496	7,146
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	5,282	61,419	3,615	6,317	3,576	1,423	4,056
Typ 2	0,200	2,131	0,142	0,376	0,005	0,168	0,045
Typ 3	1,758	21,654	1,233	2,212	5,735	1,400	6,496

Algoritmus lze ještě dále vylepšit tak, že před přechodem na algoritmus využívající převrácení místo klasické záměny bloků zaměníme bloky tak, aby blok, který by se následně musel převracet, byl již převrácen. To lze uskutečnit např. postupnou křížovou záměnou prvků od začátku s prvky od konce. Tím se ušetří jedno převrácení pole ze tří.

Po této změně ještě jednou změříme nejvhodnější hranici pro přepnutí algoritmu. To je znázorněno na obrázku 3.10. Je zde vidět, že nejvhodnější hranice se posunula na číslo 512. Konečně časy dosažené po této změně jsou v tabulce 3.17. Tento algoritmus se ukázal jako jednoznačně nejlepší v běžném případě. Výjimku tvoří situace řazení seřazených posloupností, kde vždy do-

chází k přímé degradaci na algoritmus využívající převrácení a nemůže tedy v porovnáním s ním přinést žádné zrychlení.



Obrázek 3.11: In-place Mergesort – doba řazení při hybridním přístupu pro danou hranici

Tabulka 3.17: In-place Mergesort – Hybridní rotace bloků

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	5,282	61,419	3,615	6,317	3,576	1,423	4,056
Typ 2	0,200	2,131	0,142	0,376	0,005	0,168	0,045
Typ 3	1,758	21,654	1,233	2,212	5,735	1,400	6,496
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	4,362	51,672	3,008	5,801	3,363	1,420	3,844
Typ 2	0,200	2,077	0,145	0,371	0,005	0,173	0,045
Typ 3	1,748	21,420	1,235	2,207	5,803	1,418	6,621

3.3.2 Straightforward Mergesort

Jak už bylo zmíněno, tento algoritmus bude využit k optimalizaci slučování s vnějším pomocným polem. Nejprve změříme výkonnost při použití algoritmu odpovídajícímu uvedenému algoritmu 2.3, kde místo přesunů prvků se prvky zaměňují. Tu lze vidět v tabulce 3.18.

Následně zkusíme uplatnit obdobu optimalizace ze sekce 3.2.1.2, tedy provádět kontrolu, zda jsme již nevyčerpali prvky jedné z částí, pouze pro tu část, z které jsme právě odebrali prvek. Ačkoliv u *out-of-place* Mergesortu vedla ke

Tabulka 3.18: In-place Mergesort – Straightforward Mergesort

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,614	18,585	1,006	2,138	2,707	2,813	3,361
Typ 2	0,389	4,483	0,272	0,845	1,588	1,589	1,985
Typ 3	0,553	6,540	0,427	0,918	2,352	1,947	2,872

zpomalení, nemusí tomu tak být v tomto případě. Jak je také vidět v tabulce 3.19, tato změna přinesla zrychlení ve většině případů.

Tabulka 3.19: In-place Mergesort – Straightforward Mergesort

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,614	18,585	1,006	2,138	2,707	2,813	3,361
Typ 2	0,389	4,483	0,272	0,845	1,588	1,589	1,985
Typ 3	0,553	6,540	0,427	0,918	2,352	1,947	2,872
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,445	16,588	1,072	1,950	2,683	2,780	3,393
Typ 2	0,355	4,086	0,257	0,726	1,571	1,581	2,003
Typ 3	0,540	6,386	0,426	0,837	2,331	1,935	2,866

Dále, pokud si uvědomíme, že na pořadí prvků ve vnitřním pomocném poli nám nezáleží, lze ušetřit jeden z přesunů při prohazování prvků. Toho docílíme tak, že na začátku vyjmeme první prvek z pomocného pole do pomocné proměnné. Poté, co na toto volné místo přesuneme odpovídající prvek, vyjmeme další prvek z pomocného pole, tentokrát ho však umístíme na původní místo prvku, který jsme právě přesunuli. Tak pokračujeme, dokud nepřesuneme poslední prvek, na jehož místo pak umístíme prvek z pomocného pole, který jsme měli uložený v pomocné proměnné. Jedná se o obdobu „floating hole“ techniky uvedené v [14].

Tato změna by neměla mít vliv na rychlost řazení elementárních prvků. Záměna těchto prvků totiž nepotřebuje externí paměť a používá pouze registry procesoru. Ve výsledku je tak v podstatě ekvivalentní v počtu přesunů prvků v paměti s touto optimalizací. Pokud se však podíváme na tabulku 3.20, kde jsou zachyceny dosažené časy po této změně, tak zjistíme, že z nějakého důvodu došlo ke zrychlení ve většině případů, výjimku tvoří pouze řazení prvků typu `char`. Důvod není zřejmý a bylo by ho nejspíše nutné hledat ve způsobu implementace nebo na straně překladače.

Důležitou poznámkou je, že tato změna nemá vliv na *in-place* vlastnost algoritmu, protože se jedná o konstantní paměť navíc vzhledem k typu řazeného objektu.

Tabulka 3.20: In-place Mergesort – Straightforward Mergesort

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,445	16,588	1,072	1,950	2,683	2,780	3,393
Typ 2	0,355	4,086	0,257	0,726	1,571	1,581	2,003
Typ 3	0,540	6,386	0,426	0,837	2,331	1,935	2,866
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,412	16,288	1,005	1,982	2,506	2,754	3,075
Typ 2	0,354	4,047	0,262	0,724	1,472	1,596	1,844
Typ 3	0,523	6,057	0,404	0,838	2,194	2,048	2,675

3.3.3 Block Mergesort

V tabulce 3.21 lze nalézt časy dosažené algoritmem, který odpovídá popisu tohoto algoritmu ze sekce 2.7. Je důležité zmínit, že původní algoritmus z [10] pro algoritmy pro slučování s vnitřním pomocným polem a *Rotation Merge* za účelem dosažení optimálního počtu přesunů a porovnání používá principy podle [15]. Tento postup je však v praxi velmi neefektivní a byl tak nahrazen klasickým slučováním pro první z algoritmů a binárním vyhledáváním pro *Rotation Merge*.

Také byl vzhledem k povaze funkce *Block Merge* k řazení polí délky 32 a méně rovnou použit algoritmus *Insertion sort*. Tato konstanta bude doladěna po aplikaci všech optimalizací.

Tabulka 3.21: In-place Mergesort – Block Mergesort

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	2,368	26,103	1,523	4,036	2,530	1,468	3,004
Typ 2	0,173	1,712	0,115	0,493	0,062	0,278	0,138
Typ 3	1,133	12,265	0,790	1,034	3,559	1,155	4,172

3.3.3.1 Extrakce pomocného pole předem

Extrakci pomocného pole b lze provést před voláním funkce Mergesort. Ačkoliv se jedná o extrakci z neseřazeného pole, která má složitost $\mathcal{O}(n \log n)$, je provedena pouze jednou a neovlivní tedy celkovou asymptotickou složitost řazení. Touto změnou je umožněno několik optimalizací. V první řadě samozřejmě změníme algoritmus tak, aby neprováděl extrakci a následnou distribuci vlastního pomocného pole, ale využil již extrahované pole. Místo samotné extrakce je provedeno pouze seřazení dostatečné délky pole pro zrcadlení pohybu w-bloků.

Další optimalizací umožněnou touto změnou je možnost použití Merge pomocí vnitřního pole pro pole, jejichž levá polovina má délku menší, nebo

rovnou délce extrahovaného pomocného pole. Také je možné změnit délku bloků d na $|b|$, tedy v případě, že $|b| > \lfloor\sqrt{|u|}\rfloor$. V opačném případě se délka bloku nastaví na $d = \lfloor\frac{|u|}{|b|}\rfloor$. Jelikož je délka extrahovaného pomocného pole známa předem, je znám předem i počet bloků a lze tak nastavit délku bloků s_1 a s_2 podle této délky, nikoliv podle její maximální možné hodnoty.

Extrakcí pomocného pole předem také zjistíme minimální počet různých prvků v poli, šlo by tedy uvažovat o použití speciálního algoritmu pro pole s např. dvěma prvky. Dále pak v případě pouze jednoho prvku samozřejmě nemá smysl řazení provádět. V této práci nejsou žádné speciální algoritmy použity.

V tabulce 3.22 lze vidět, že tyto optimalizace přinesly výrazné zrychlení ve většině případů, včetně řazení pole náhodně uspořádaných prvků typu `char`, kde neexistuje dostatečný počet různých prvků. Výjimku pak tvoří některé případy již seřazených posloupností, kde se nejspíše projeví fakt, že při distribuci pomocného pole zpět je binárně prohledáváno celé pole a ne jen část první poloviny pole.

Tabulka 3.22: In-place Mergesort – Extrakce pomocného pole předem

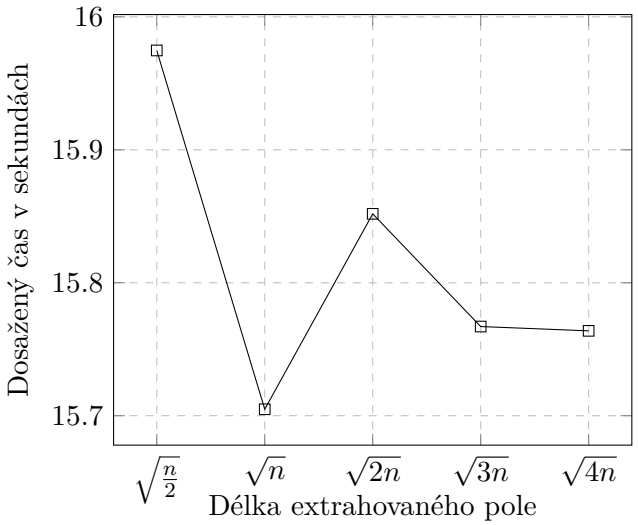
Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	2,368	26,103	1,523	4,036	2,530	1,468	3,004
Typ 2	0,173	1,712	0,115	0,493	0,062	0,278	0,138
Typ 3	1,133	12,265	0,790	1,034	3,559	1,155	4,172
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,545	17,566	1,023	3,192	2,252	1,214	2,630
Typ 2	0,125	1,415	0,084	0,611	0,310	0,281	0,393
Typ 3	0,739	8,586	0,542	1,088	3,303	1,102	3,879

3.3.3.2 Small Buffer Merge

Tato optimalizace nahrazuje použití algoritmů *Rotation Merge* a *Block Rotation Merge* v algortimu *Block Merge* novým algoritmem, který využívá dostupné vnitřní pomocné pole, které však není dostatečně velké. Probíhá stejným způsobem jako algoritmus *Block Rotation Merge* ze sekce 2.7.3 s velikostí bloků $d = |b|$, kde b je vnitřní pomocné pole. Jediný rozdíl je ten, že na sloučení dvojic $u_i v_i$ používáme algoritmus pro sloučení s pomocí vnitřního pomocného pole, proto je velikost bloků $d = |b|$.

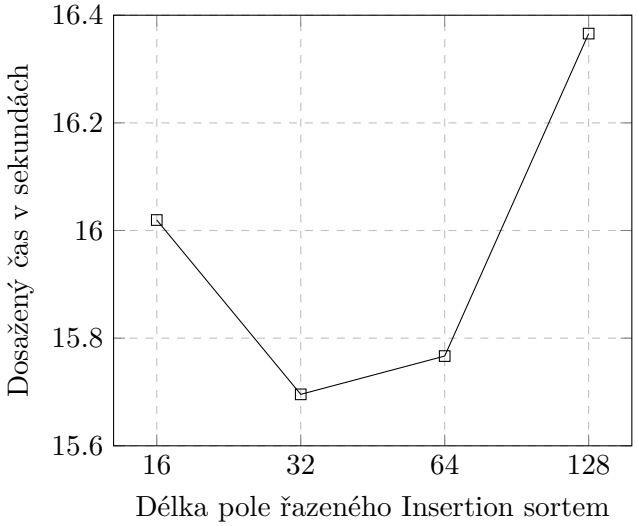
Jelikož se jedná o poslední optimalizaci tohoto algoritmu, provedem nejprve experimentální volbu některých konstant. První z nich bude délka extrahovaného pomocného pole. Po předchozí optimalizaci je totiž snadné změnit délku extrahovaného pole bez dalších zásahů do algoritmu. Na obrázku 3.12 jsou zobrazeny časy dosažené pro dané délky extrahovaného pole, kde n značí

délku celého pole. Jako nejrychlejší se ukázalo extrahovat pole o délce $\sqrt{n}$, tedy o něco větší než minimální délka.



Obrázek 3.12: In-place Mergesort – doba řazení pro danou délku extrahovaného pole

Dále přeměříme délku pole řazenou Insertion sortem. Dosažené časy lze vidět na obrázku 3.13. Překvapivě se i přes nižší efektivitu *in-place* řazení v porovnání s *out-of-place* ukázala jako nejrychlejší stejná hodnota, tj. 32.



Obrázek 3.13: In-place Mergesort – doba řazení při použití Insertion sortu pro danou délku pole

Konečně čas dosažený po poslední optimalizaci a při použití těchto konstant lze vidět v tabulce 3.23. Ke zrychlení došlo ve všech náhodně uspořádaných případech. K mírnému zpomalení došlo v některých speciálních případech, tedy již seřazených a obráceně seřazených posloupností. Zpomalení v těchto případech je pochopitelné, protože operace prováděné nahrazenými algoritmy jsou v těchto případech efektivnější. *Rotation Merge* totiž pro již seřazené pole neprovádí žádné přesuny a pro opačně seřazené provede pouze 1 přesun a 1 porovnání.

Tabulka 3.23: In-place Mergesort – Small Buffer Merge

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,545	17,566	1,023	3,192	2,252	1,214	2,630
Typ 2	0,125	1,415	0,084	0,611	0,310	0,281	0,393
Typ 3	0,739	8,586	0,542	1,088	3,303	1,102	3,879
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,375	15,692	0,969	3,015	2,056	1,178	2,363
Typ 2	0,130	1,455	0,089	0,611	0,341	0,297	0,431
Typ 3	0,650	7,558	0,458	1,095	2,807	1,288	3,286

Kapitola 4

Paralelizace

Tato kapitola obsahuje popis paralelizace algoritmů uvedených v kapitole 2. Dále obsahuje časy běhů algoritmů naměřené v průběhu tohoto procesu.

4.1 Out-of-place Mergesort

Nejprve se zaměříme na paralelizaci *out-of-place Mergesortu*.

4.1.1 Základní algoritmus

Základní principy paralelizace *Mergesortu* jsou čerpány z [16]. Při paralelizaci algoritmu Mergesort je důležité si uvědomit, že pouhá paralelizace funkce Mergesort nepřinese největší možné zrychlení, neboť v nejvyšších úrovních rekurze může najednou pracovat pouze omezený počet vláken. Jelikož se jedná o slučování nejdelších polí, není tento čas zanedbatelný. Pro dosažení maximální možné paralelizace je tedy nutné provést i paralelizaci funkce Merge.

4.1.1.1 Paralelizace funkce Mergesort

Paralelizace funkce Mergesort je přímočará a spočívá v rozvětvení v okamžiku rekurze, kdy každou větev rekurze může provádět jiné vlákno, neboť pracují s jinými částmi pole. Jakmile se obě větve provedou, provede se volání funkce Merge. Jelikož každá větev může trvat jinou dobu, není vhodné provádět dělení jen dokud každé vlákno nemá alespoň jednu větev, ale tak, aby průměrně na 1 vlákno připadalo více podúloh. Zároveň není dobré vytvářet maximální možný počet podúloh, protože by se výrazně zvýšila režie.

Lze tedy provádět větvení např. do určité délky pole, nebo na předem daný počet podúloh, kterým může být nějaký násobek počtu vláken. Daný počet podúloh je také vhodné zaokrouhlit na nejbližší vyšší mocninu dvou, pro lepší rozdělení práce vzhledem k povaze funkce Mergesort, kdy počet úloh ve stejné hloubce rekurze je vždy mocnina dvou.

4.1.1.2 Paralelizace funkce Merge

Paralelizace funkce Merge spočívá v rozdělení dvou částí vstupního pole na více částí, jejichž slučování lze provádět nezávisle a tedy paralelně, přičemž výsledek zůstane identický. To je uskutečněno rozdělením první části pole na daný počet částí. Druhé pole se pak rozdělí tak, aby poslední prvky jednotlivých částí byly posledními prvky v celém bloku, které jsou menší než poslední prvky korespondujících částí první poloviny pole. Pravé části tak vždy obsahují prvky zaručeně menší než poslední prvky odpovídajících levých částí a jejich nezávislé slučování je tedy stabilní.

Jelikož pravé části budou různých délek, bude velmi často i jejich slučování trvat různou dobu. Opět by tedy počet částí a tedy podúloh neměl být přímo roven počtu vláken, ale nějakému jeho násobku, podobně jako u paralelizace funkce Mergesort. Nabízí se volit stejný počet jako pro funkci Mergesort. Každá úroveň rekurze je tak dělená do přesně daného počtu podúloh, kterým je násobek počtu vláken, což by mělo zajistit dostatečně vyvážené zatížení jednotlivých vláken.

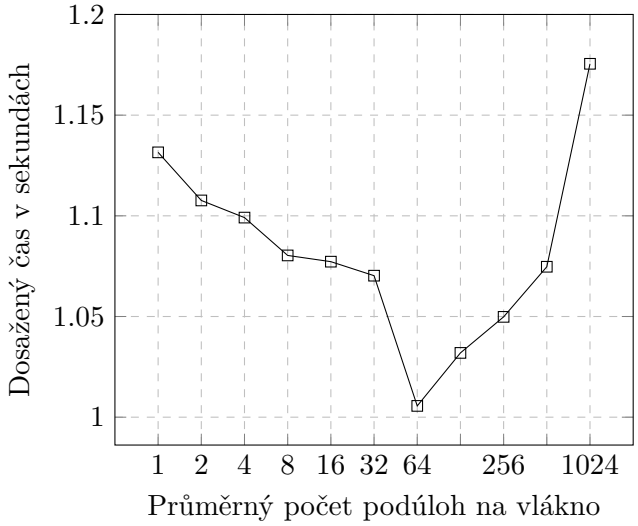
Dělící prvky lze efektivně najít binárním vyhledáváním v čase $\mathcal{O}(k \log n)$, kde k je násobek počtu vláken, lze ho tedy považovat za konstantu a předpokládat, že $k \ll n$. Ve výsledku tak tato operace nijak neovlivní celkovou asymptotickou časovou složitost algoritmu. Hledání těchto prvků lze uskutečnit paralelně. V této práci se však k přesunu prvků využívá způsobu, který nezaručuje zachování prvku na předchozím místě. Při paralelním hledání by tedy mohlo dojít k situaci, že je hledáno mezi prvky, které už nejsou platné. Nebo by bylo možné nejprve vyhledat dělící prvky a uložit je do nějaké pomocné tabulky a teprve poté začít slučovat. To by však zvýšilo režii a využitou paměť, navíc by také nebylo umožněno vyvážení zátěže, které umožňuje následující postup.

Jelikož je počet podúloh zaokrouhlen na mocninu 2, je hledání dělících prvků částečně paralelizováno následujícím způsobem. Levá část se rozdělí na polovinu a v pravé části se nalezne odpovídající dělící prvek. Poté se rekurzivně provádí to samé pro obě poloviny až do hloubky dostatečné pro vytvoření daného počtu podúloh. Větve této rekurze jsou prováděny paralelně. Tento způsob také umožňuje střídat roli levé a pravé poloviny podle toho, která je momentálně delší. Pokud totiž budeme přesně na polovinu dělit větší z částí a přizpůsobovat tu menší z částí, ve většině případů tak dojde k vyváženějšímu dělení, protože minimální počet prvků po dělení bude logicky větší a případná nevyváženost tak menší.

4.1.1.3 Měření

Nejprve bylo provedeno experimentální určení poměru počtu podúloh k počtu vláken. Obrázek 4.1 zachycuje dosažené časy pro dané poměry. Vzhledem k zaokrouhlování na nejbližší vyšší mocninu dvou byly jako poměry voleny pouze

mocniny dvou. Jako nejlepší se pak ukázal poměr 64. Na grafu je znatelný skok právě v tomto bodě. Jednalo se o náhodně výjimečně dobrý dosažený čas. Další měření dosáhly času o něco pomalejšího než poměr 128, ale pro tuto konstantu bylo zaznamenáno výraznější zpomalení v případech rychlejších řazení, tedy režie se zřejmě stávala příliš výraznou. Jako výsledný poměr byla tak zvolena konstanta 64.



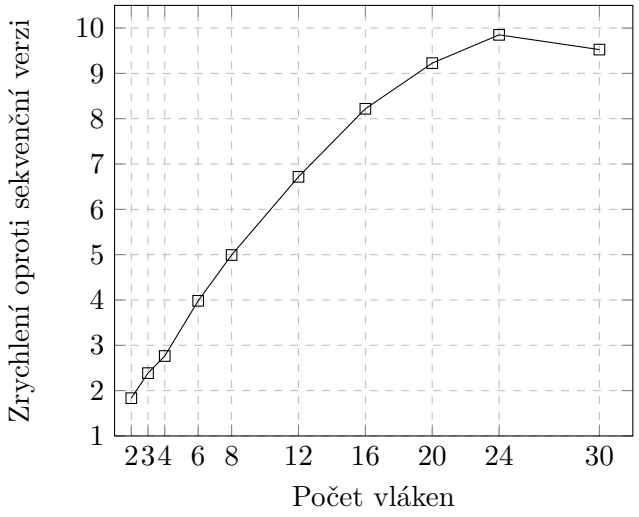
Obrázek 4.1: Out-of-place Mergesort – doba řazení při daném počtu podúloh

Na obrázku 4.2 lze vidět dosažené zrychlení pro různé množství vláken při použití konstanty 64 pro počet podúloh na vlákno. Zrychlení má přibližně lineární charakter vzhledem k počtu vláken. Ke zlomu dochází při použití více než 12 vláken. Jedním z důvodů může být existence pouhých 12 fyzických jader. Ty podporují technologii *Hyper-Threading*, která umožňuje běh až 2 vláken najednou na jednom fyzickém jádru. Tento souběh 2 vláken na jednom jádru však není ideální a je méně efektivní než by bylo 24 fyzických jader. Druhým důvodem pak samozřejmě může být omezená paralelizace funkce Merge.

Také bylo provedeno měření pro 30 vláken pro ověření, že skutečně dochází k využívání technologie *Hyper-Threading* a ke zrychlení nedošlo z důvodu vyplnění nečinného času jednoho vlákna druhým nebo kvůli implicitně většímu počtu podúloh. Vzhledem k dosaženému času se dá předpokládat, že skutečně dochází k běhu 2 vláken na 1 fyzickém jádru.

Konečně v tabulce 4.1 jsou zaznamenány časy dosažené při použití 24 vláken. Při porovnání s tabulkou sekvenční verze si lze všimnout, že pro řazení která trvala kratší dobu, je zrychlení o něco nižší. To není nic překvapivého, vzhledem k tomu že režie pro práci s vlákny zůstává přibližně stejná a projeví se tedy výrazněji při kratším řazení.

Dále stojí za zmínku časy dosažené při řazení objektů typu `BigObject<4096>`



Obrázek 4.2: Out-of-place Mergesort – zrychlení při daném počtu vláken

a `BigSlow<4096>`. Jejich zrychlení je výrazně menší oproti ostatním. Faktem je, že stejného času bylo dosaženo už při použití 6 vláken. Pravděpodobným důvodem je tedy omezení ze strany propustnosti paměti. Podobně je na tom pak i řazení objektů typu `SlowCompare`, které přestalo zaznamenávat zrychlení pro více než 12 vláken, nejspíše ze stejného důvodu.

Tabulka 4.1: Out-of-place Mergesort – paralelní varianta s 24 vlákny

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,117	1,006	0,098	0,182	0,501	0,252	0,857
Typ 2	0,065	0,566	0,067	0,080	0,487	0,089	0,784
Typ 3	0,067	0,627	0,068	0,088	0,517	0,100	0,528

4.1.2 Out-of-place Mergesort s polovičním pomocným polem

Funkce *Mergesort* navrhované v předchozích optimalizacích nelze paralelizovat, protože jednotlivé větve pracují se stejnými částmi pole. Výjimku tvoří druhá varianta ze sekce 3.2.2.5 a její obdoba ze sekce 3.2.2.6. Ta však k jednomu z řazení používá první variantu, kterou opět nelze paralelizovat. Paralelizace této varianty tedy bude uskutečněna pomocí paralelizace klasické verze ze sekce 4.1.1. Bude tedy využívat pomocné pole o délce n. Jediným rozdílem pak bude využití nejlepší z variant *Mergesortu* s polovičním pomocným polem v sekvenční části, tedy variantu ze sekce 3.2.2.2.

4.1.2.1 Měření

V tabulce 4.2 jsou pak zaznamenány časy pro 24 vláken. V porovnání s paralelizací klasické verze lze vzhledem k tomu, že kromě času dosaženého pro Data 2, který byl mimořádný a nepodařilo se ho zopakovat, předčila tato varianta předchozí ve většině případů, označit za úspěšnější a prohlásit tedy tuto variantu za lepší.

Tabulka 4.2: Out-of-place Mergesort – Paralelní verze

Před	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,117	1,006	0,098	0,182	0,501	0,252	0,857
Typ 2	0,065	0,566	0,067	0,080	0,487	0,089	0,784
Typ 3	0,067	0,627	0,068	0,088	0,517	0,100	0,528
Po	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,115	1,018	0,098	0,162	0,496	0,258	0,861
Typ 2	0,062	0,573	0,066	0,078	0,480	0,092	0,511
Typ 3	0,071	0,610	0,068	0,092	0,503	0,100	0,741

4.2 In-place Mergesort

Nyní popíšeme paralelizaci *in-place Mergesortu*.

4.2.1 RecMerge

RecMerge lze snadno paralelizovat paralelním prováděním různých větví rekurze, protože pracují s odlišnými daty. Podobně jako Mergesort však tato paralelizace neumožní využití všech vláken v nejvyšších úrovních. Rotaci bloků lze provádět paralelně, jelikož je to však velmi rychlá operace, ve většině případů se ji nevyplatí paralelizovat. Samotná výkonnost paralelní verze tohoto algoritmu pak není důležitá a nebude tedy měřena. Tento princip je uveden z důvodu využití v následujících algoritmech.

4.2.2 Block Mergesort

Stejně jako v předchozích případech spočívá paralelizace tohoto algoritmu v paralelizaci funkce *Mergesort* a funkce *Merge*. Paralelizaci funkce *Mergesort* lehce komplikuje extrakce pomocného pole předem. Tuto extrakci nelze efektivně provádět paralelně, navíc každé vlákno potřebuje své vlastní pomocné pole. Paralelizace je z těchto důvodů prováděna následujícím způsobem. Opět je každá větev rekurze rozdělena na daný počet podúloh, který odpovídá nějakému násobku počtu vláken, jako v *out-of-place* verzi. Jakmile se pak přejde na sekvenční část, je provedena extrakce pomocného pole, které je následně

distribuováno zpět do pole po skončení sekvenční části. Pole v paralelně prováděné části jsou řazena algoritmem bez této optimalizace.

Pro paralelizaci funkce *Merge* si vezmeme inspiraci z paralelizace v sekci 4.2.1. Tedy rozdělíme větší z částí pole na polovinu a ve druhé části nalezneme odpovídající dělící prvek. Poté provedeme rotaci tak, že vzniknou dvě dvojice bloků, jejichž nezávislým sloučením vznikne seřazená posloupnost. Toto dělení provádíme rekurzivně, dokud nevznikne dostatečný počet podúloh. Jednotlivé větve lze provádět paralelně. Poté bloky sloučíme pomocí algoritmu *Block Merge*.

Problémem však zůstává velká výsledná nevyváženost podúloh. K lepšímu vyvážení využijeme stejného triku jako v *out-of-place* verzi, tedy budeme střídat roli levé a pravé poloviny pole podle toho, která je právě teď delší. Tato změna však povede k situacím, kdy levá polovina bude příliš krátká na to, aby z ní bylo extrahováno dostatečně dlouhé pole vzhledem k délce pravé poloviny.

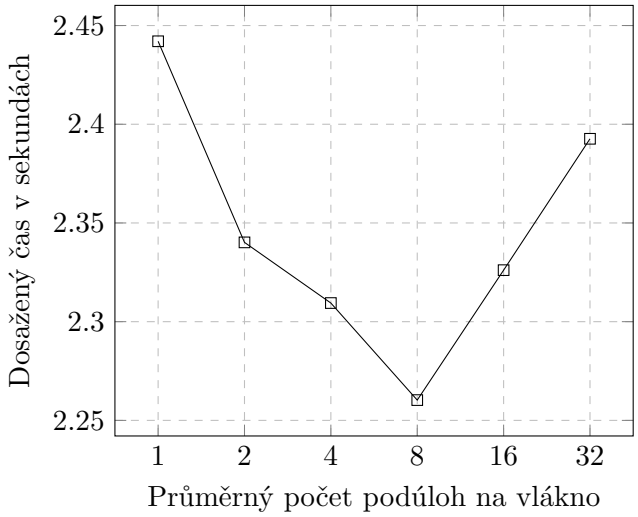
Nejvhodnější by tedy bylo vytvořit algoritmus, který by pracoval zrcadlově v porovnání s použitým algoritmem. To je však mimo rámec této práce a na vyřešení tohoto problému bylo zvoleno jednodušší řešení, které vyžaduje minimální zásahy do původního algoritmu. Konkrétně se jedná o extrakci pomocného pole z pravé poloviny v případě, kdy délka pravé poloviny je větší, než délka levé poloviny. Všechny ostatní role levé a pravé poloviny v rámci algoritmu zůstanou identické.

Tuto extrakci je pak pro zachování stability nutné provádět od konce pravé poloviny, stejně jako následnou distribuci zpět do pole. Je tedy nutné vytvořit alternativní algoritmy pro *Rotation Merge* a extrakci pomocného pole, které pracují zrcadlově. Dále nastanou malé změny ke konci algoritmu, kde figuruje pomocné pole b v jedné z rotací bloků. Konkrétně je rotováno z bw_1v_1' na $v_1'bw_1$. Jelikož se zde b nenachází, bude samozřejmě rotováno pouze z w_1v_1' na $v_1'w_1$.

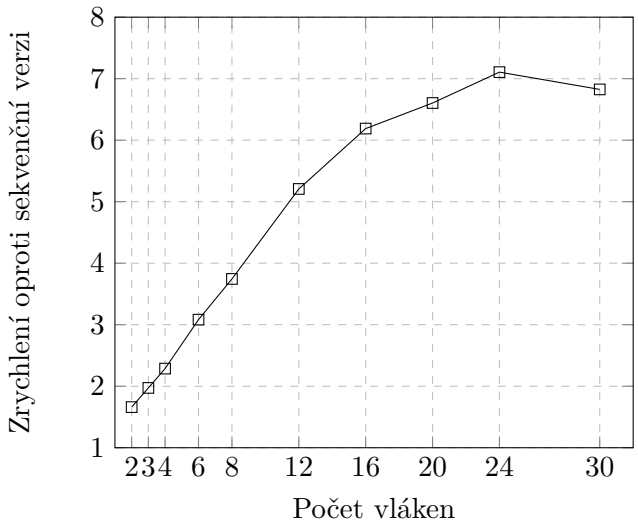
4.2.2.1 Měření

Nyní je na čase experimentálně zvolit vhodný počet podúloh na 1 vlákno. Na obrázku 4.3 lze vidět časy dosažené pro různé konstanty. Jako nejeefektivnější se jeví hodnota 8. Zrychlení pro různý počet vláken při použití této konstanty lze vidět na obrázku 4.4. Zrychlení se opět jeví lineární pro malý počet vláken, pro více než 12 vláken však nastává větší zlom, zřejmě kvůli nižší efektivitě paralelizace *in-place* slučování, které vyžaduje nejen vyhledání zlomových prvků, ale také rotaci odpovídajících bloků. Dále bylo opět provedeno měření pro 30 vláken pro ověření, že zrychlení bylo docíleno díky technologii Hyper-Threading a nikoliv kvůli vyplnění nečinnosti jednoho vlákna druhým nebo větším počtem celkových podúloh.

Konečně pak v tabulce 4.3 lze vidět finální časy dosažené při použití tohoto paralelního *in-place* algoritmu. Lze si všimnout že pro objekty typu `BigObject` a `BigSlow` došlo k menšímu zrychlení. U těchto objektů opět nedošlo k žád-



Obrázek 4.3: In-place Mergesort – doba řazení při daném k-násobku podúloh



Obrázek 4.4: In-place Mergesort – zrychlení paralelní verze při daném počtu vláken

ném zrychlení pro více než 6 vláken. Pro objekt `SlowCompare` naopak docházelo k zrychlení až pro 20 vláken, což je v porovnání s paralelní *out-of-place* variantou lepší.

Tabulka 4.3: In-place Mergesort – Paralelní verze

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,212	2,260	0,191	0,468	1,364	0,326	1,471
Typ 2	0,022	0,221	0,016	0,162	0,049	0,061	0,066
Typ 3	0,167	1,885	0,165	0,274	1,460	0,432	1,765

Kapitola 5

Hybridní algoritmus

Tato kapitola obsahuje popis autorem navrženého hybridního algoritmu, který využívá *in-place* a *out-of-place Mergesort* popsaný v předchozích kapitolách této práce.

5.1 Srovnání in-place a out-of-place Mergesortu

Out-of-place Mergesort je ve většině případů rychlejší než jeho *in-place* varianty. Výjimku tvoří pouze speciální typy vstupních posloupností – již seřazené, pro které je *in-place* rychlejší. Tuto vlastnost však nelze obecně predikovat. Proto bude přístup hybridního algoritmu navržen z hlediska velikosti poskytnutého pomocného pole. Nejprve bude navržena sekvenční verze a následně paralelní verze.

5.2 Sekvenční verze

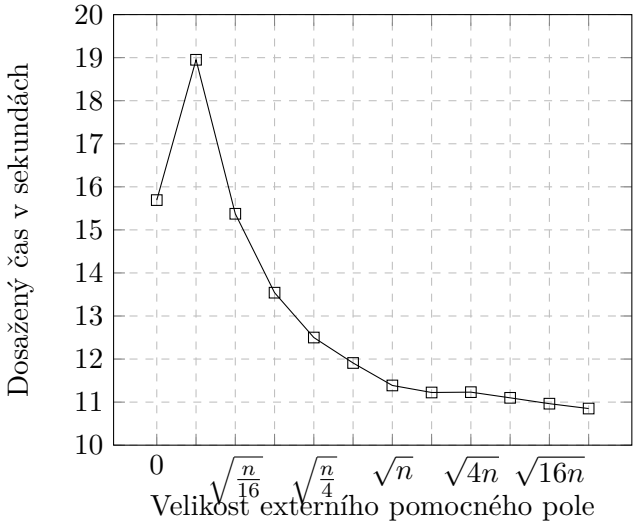
Sekvenční verze je z hlediska návrhu o něco jednodušší, není totiž třeba uvažovat o dělení poskytnutého pomocného pole mezi jednotlivými vlákny. Abychom se vyhnuli konfliktu s označením vnitřního pomocného pole b, označíme vnější pomocné pole a.

Situace pro $|a| \geq \lceil \frac{n}{2} \rceil$ a $|a| = 0$ je jednoznačná. V první z nich použijeme nejlepší sekvenční *out-of-place* algoritmus, tedy algoritmus ze sekce 3.2.2.2. V případě žádného externího pomocného pole samozřejmě použijeme nejlepší *in-place* variantu, tedy variantu ze sekce 3.3.3.2. Otázkou pak zůstává, co dělat v ostatních případech. Pro tyto případy tedy navrhneme speciální algoritmus, který bude vycházet z algoritmu *Block Mergesort*.

Tento algoritmus bude postupovat obdobným způsobem s malými změnami. První změnou je, že jakmile se v rámci rekurze dostaneme k řazení pole, pro které platí $|a| \geq \lceil \frac{n}{2} \rceil$, přepneme na *out-of-place* algoritmus. Další změnou bude nahrazení slučování s vnitřním pomocným polem slučováním

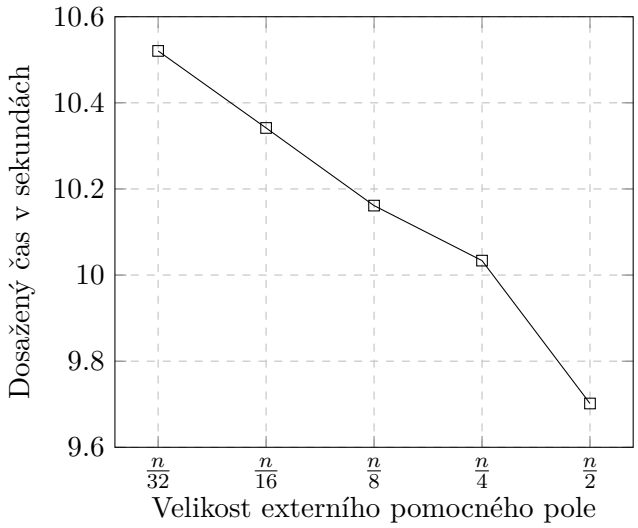
s vnějším pomocným polem v rámci *in-place* algoritmu. Po této změně bude vnitřní pomocné pole využito pouze při reorganizaci bloků. Jeho maximální potřebná délka tak bude $|b| = \frac{n}{2|a|}$. Zároveň tuto délku omezíme tak, aby $|b| \leq \sqrt{n}$. Důležité je si uvědomit, že během reorganizace bloků se v pomocném poli b imituje pohyb w-bloků a jelikož je na konci této operace zachováno pořadí těchto bloků, zůstává pomocné pole b stále seřazené a není ho nutné v průběhu algoritmu znovu řadit.

Na obrázcích 5.1 a 5.2 lze vidět dosažený čas pro různou délku poskytnutého pomocného pole. Z prvního grafu je patrné, že pro $|a| < \sqrt{\frac{n}{16}}$ došlo ke zpomalení. Provedeme tedy finální úpravu algoritmu a to tak, aby pro $|a| < \sqrt{\frac{n}{16}}$ byla použita *in-place* varianta. Dále si lze všimnout že velké části zrychlení je dosaženo už při poměrně malém pomocném poli a. Konkrétně pak např. poslední hodnota na prvním grafu označuje čas dosažený s přibližně 50krát menším pomocným polem než první hodnota druhého grafu a dosažený čas je přibližně pouze o 3 % pomalejší. Z druhého grafu je také patrné, že přesun z hybridního algoritmu na čistě *out-of-place* algoritmus je téměř plynulý.



Obrázek 5.1: Hybridní algoritmus – doba řazení při dané velikosti externího pomocného pole

V tabulce 5.1 jsou zaznamenány časy dosažené pro reprezentativní hodnotu $|a| = \sqrt{n}$.



Obrázek 5.2: Hybridní algoritmus – doba řazení při dané velikosti externího pomocného pole

Tabulka 5.1: Hybridní algoritmus – sekvenční verze

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,993	11,387	0,725	2,988	1,576	1,102	1,895
Typ 2	0,206	2,298	0,138	0,782	0,202	0,238	0,266
Typ 3	0,677	8,035	0,558	1,371	2,155	1,132	2,559

5.3 Paralelní verze

Návrh paralelní verze začneme stejně jako návrh sekvenční verze. Při $|a| = n$ bude použit nejlepší *out-of-place* algoritmus, tedy algoritmus ze sekce 4.1.2. Při $|a| = 0$ bude použit optimalizovaný paralelní algoritmus *Block Mergesort* ze sekce 4.2.2 beze změn. Nyní opět navrhneme hybridní algoritmus pro ostatní případy na základě *in-place* algoritmu.

Paralelizace funkce Mergesort bude opět provedena paralelním prováděním jednotlivých větví. Jakmile dojde na sekvenční část, použijeme sekvenční verzi hybridního algoritmu. Poté je opět zapotřebí paralelizovat i funkci Merge. K tomu využijeme paralelizace funkce *in-place* Merge, tedy variantu s vyhledáváním dělících prvků a následnou rotací odpovídajících bloků. Poté provedeme obdobu Merge ze sekvenční hybridní verze, s tím rozdílem, že nebudou uplatněny optimalizace ze sekce 3.3.3.1, protože je zapotřebí extrahovat pomocné pole před tímto slučováním a následně distribuovat ho zpět.

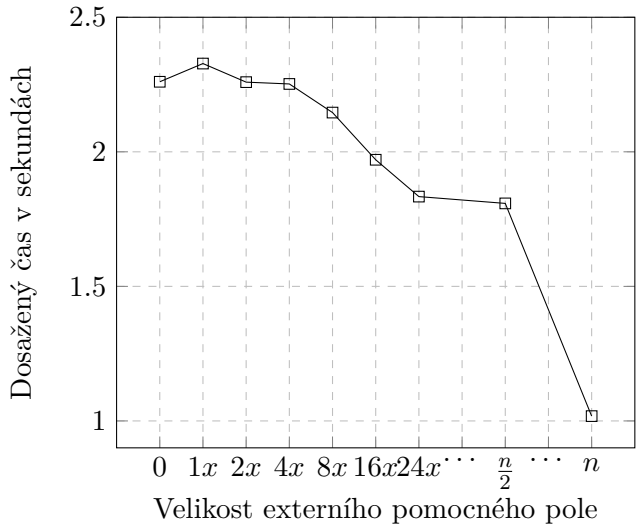
Hlavním problémem paralelní verze je nutnost sdílení pomocného pole mezi více vlákny. Abychom mohli využívat co nejmenší pomocné pole, povolíme, aby pouze některá z vláken využívala externí pomocná pole, za cenu nevyvá-

ženého výkonu jednotlivých vláken. Určíme tedy minimální délku pomocného pole pro 1 vlákno, tuto délku označíme x. Z důvodu dostatečně významného dopadu budeme požadovat, aby při řazení pole o délce n bylo $x \geq \sqrt{\frac{n}{2}}$.

Po rozdělení na kt (zaokrouhleno na nejbližší vyšší mocninu 2) podúloh, kde k je počet podúloh na vlákno a t je počet vláken, bychom požadovali pomocné pole o velikosti alespoň $x = \sqrt{\frac{n}{2kt}}$. Jelikož však délka pole pro paralelní Merge nebude vždy přesně vyvážena, zvýšíme tento požadavek na $x = \sqrt{\frac{2n}{kt}}$. Nyní provedeme měření pro různé délky pomocného pole a.

Výsledky měření jsou vyneseny na obrázku 5.3. Při dostupnosti externího pole pouze pro jedno vlákno došlo k mírnému zpomalení, zřejmě kvůli nevyváženosti výkonnosti vláken. Výraznější zrychlení bylo dosaženo až když externí pomocné pole mělo k dispozici 8 a více vláken. Jakmile však bylo k dispozici externí pomocné pole pro všechny vlákna, bylo dosaženo téměř maximálního zrychlení a poskytnutí většího externího pole nemělo téměř žádný význam. Oproti sekvenční verzi je tedy přechod z hybridního přístupu na *out-of-place* algoritmus velkým skokem.

Důvodů k tomuto skoku je několik. Velkou nevýhodou je používání rotací při paralelizaci funkci Merge. Dále pak nutnost dělení pomocného pole mezi vlákny. Paralelní verze také provádí mnohem více extrakcí vnitřního pomocného pole a tedy následných distribucí. Nicméně ke zrychlení oproti *in-place* verzi dochází a tedy i tento algoritmus má svůj význam.



Obrázek 5.3: Hybridní algoritmus – doba paralelního řazení při dané velikosti externího pomocného pole

V tabulce 5.2 jsou zaznamenány časy dosažené pro reprezentativní hodnotu $|a| = 24x$.

Tabulka 5.2: Hybridní algoritmus – paralelní verze

Vstup	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,183	1,834	0,149	0,333	1,294	0,324	1,458
Typ 2	0,171	1,574	0,117	0,126	0,060	0,363	0,134
Typ 3	0,285	2,600	0,208	0,298	1,501	0,511	1,686

5.4 Stabilita hybridního algoritmu

Sekvenční i paralelní verze hybridního algoritmu využívají popsané principy stabilního *in-place* a *out-of-place* Mergesortu. Jsou kombinovány způsobem, který nijak neovlivňuje stabilitu. Pokud je totiž použit stabilní algoritmus na seřazení 2 částí pole, které jsou následně stabilně sloučeny, nezáleží na tom, o jaký algoritmus se jedná, výsledek bude vždy stabilní.

Dále je provedena změna v *in-place* algoritmu pro slučování při hybridním přístupu, kdy slučování pomocí vnitřního pomocného pole je nahrazeno slučováním pomocí vnějšího pomocného pole. Toto slučování je samozřejmě také stabilní. Jelikož vnitřní pomocné pole obsahuje rozdílné prvky a jeho poloha se po dokončení slučování nijak nemění, nemůže tato změna nijak ovlivnit stabilitu.

Kapitola 6

Vyhodnocení výkonnosti

Tato kapitola obsahuje porovnání výkonnosti několika existujících implementací s odpovídajícími implementacemi algoritmů vytvořených v rámci této práce.

6.1 std::stable_sort

Tato implementace je součástí standardní knihovny jazyka *C++* a je tedy součástí použitého překladače *g++* verze 4.8.5. Jedná se o implementaci stabilního *out-of-place* řazení. Dosažené časy při použití tohoto algoritmu jsou zaznamenány v tabulce 6.1 pod názvem Alg. 1. Alg. 2 je pak sekvenční hybridní algoritmus z této práce využívající pomocného pole maximální délky. Algoritmus implementovaný v rámci této práce je rychlejší ve všech měřených případech.

Tabulka 6.1: std::stable_sort

Alg. 1	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,938	10,750	0,674	1,860	1,557	1,288	1,774
Typ 2	0,530	6,209	0,443	0,928	1,634	1,057	1,672
Typ 3	0,562	6,600	0,493	0,957	1,770	1,015	1,773
Alg. 2	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,849	9,705	0,597	1,750	0,932	0,955	1,158
Typ 2	0,389	4,420	0,264	0,788	0,502	0,511	0,653
Typ 3	0,468	5,636	0,374	0,920	1,015	0,876	1,222

6.2 ___gnu_parallel::stable_sort

Tato implementace je součástí standardní knihovny GNU libstdc++, což je implementace standardní knihovny jazyka C++, která je součástí použitého

překladače *g++* verze 4.8.5. Jedná se o implementaci paralelního stabilního *out-of-place* řazení. V tabulce 6.2 jsou zaznamenány časy dosažené tímto algoritmem při použití 24 vláken. Algoritmus je označen jako Alg. 1. Dále jsou v tabulce časy dosažené pomocí Alg. 2, kterým je paralelní hybridní algoritmus z této práce využívající pomocné pole maximální délky a 24 vláken. Algoritmus `__gnu_parallel::stable_sort` je ve většině případů o něco pomalejší, výjimku tvoří pouze řazení objektů typu `BigObject` a `BigSlow`. Naopak při řazení objektů typu `SlowCompare` je tento algoritmus více než 2x pomalejší. Zřejmě tedy provádí více porovnání a méně přesunů než náš algoritmus.

Dále je důležité zmínit, že pro tento algoritmus nedošlo ke zrychlení při použití 24 vláken oproti 12 vláknům. Lze tedy předpokládat, že nedokáže využít technologii *Hyper-Threading*. Při použití 12 vláken je pak rychlejší než náš algoritmus, výjimkou je pouze řazení objektů typu `SlowCompare`.

Tabulka 6.2: ___gnu_parallel::stable_sort

Alg. 1	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,156	1,381	0,109	0,184	0,481	0,553	0,611
Typ 2	0,053	0,889	0,045	0,096	0,427	0,158	0,510
Typ 3	0,058	0,847	0,047	0,116	0,434	0,160	0,530
Alg. 2	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,114	1,041	0,094	0,179	0,484	0,257	0,834
Typ 2	0,065	0,538	0,066	0,075	0,462	0,103	0,757
Typ 3	0,066	0,609	0,065	0,093	0,404	0,105	0,782

6.3 Paralelní stabilní řazení v C++ využívající OpenMP

Jedná se o paralelní implementaci Mergesortu, dostupnou z [17], konkrétně o OpenMP variantu. Výsledky dosažené tímto algoritmem jsou zaznamenány v tabulce 6.3, kde je označen jako Alg. 1. V porovnání s časy dosaženými naším paralelním hybridním algoritmem, využívající pomocné pole maximální délky a 24 vláken, který je zde označen jako Alg. 2, si tento algoritmus vedl při řazení elementárních datových typů výrazně hůře. Při řazení objektů typu `BigObject` a `SlowCompare` dosáhl vyrovnaných časů a pro `BigSlow` předčil náš algoritmus.

6.4 Wikisort

Wikisort je implementace sekvenčního *in-place* Mergesortu, dostupná z [18]. Vychází ze stejného algoritmu pro slučování jako *in-place* algoritmus v této práci, tedy z [10]. Časy dosažené při použití tohoto algoritmu se nacházejí

Tabulka 6.3: Paralelní stabilní řazení v C++ využívající OpenMP

Alg. 1	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,261	2,150	0,136	0,614	0,477	0,253	0,584
Typ 2	0,316	3,094	0,168	0,671	0,403	0,082	0,532
Typ 3	0,311	3,156	0,164	0,684	0,433	0,078	0,549
Alg. 2	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	0,114	1,041	0,094	0,179	0,484	0,257	0,834
Typ 2	0,065	0,538	0,066	0,075	0,462	0,103	0,757
Typ 3	0,066	0,609	0,065	0,093	0,404	0,105	0,782

v tabulce 6.4 pod názvem Alg. 1. V porovnání s časy dosaženými naším sekvenčním hybridním algoritmem bez externího pomocného pole, který je zde označen jako Alg. 2, byl při řazení náhodných vstupů spíše pomalejší. Výjimku tvoří řazení objektů typu `char`, `BigObject` a `BigSlow`. Lze tedy předpokládat, že tento algoritmus provádí méně přesunů a vede si lépe při řazení polí, kde existuje velmi málo rozdílných hodnot.

Dále je tento algoritmus adaptivní, konkrétně před slučováním kontroluje, zda bloky už nejsou ve správném pořadí, nebo by byly při pouhé záměně bloků. Předčil tedy výrazně náš algoritmus při řazení již seřazených a opačně seřazených polí. Dále je vhodné uvést, že tento algoritmus využívá externí pomocné pole fixní délky 512.

Tabulka 6.4: Wikisort

Alg. 1	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,540	17,910	1,103	1,982	1,681	1,637	2,123
Typ 2	0,027	0,306	0,022	0,114	0,002	0,151	0,041
Typ 3	0,205	2,116	0,184	0,519	1,034	0,389	1,278
Alg. 2	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7
Typ 1	1,442	16,489	0,950	3,067	2,055	1,180	2,385
Typ 2	0,123	1,408	0,087	0,577	0,342	0,298	0,423
Typ 3	0,624	7,142	0,458	1,066	2,812	1,295	3,281

6.5 Možnosti budoucího rozšíření této práce

Jako další oblasti výzkumu se nabízí vhodnější využití nejkratších délek pomocného pole sekvenční varianty hybridního algoritmu, kde pro velmi krátká pomocná pole se používá *in-place* varianta Mergesortu, protože hybridní přístup vedl ke zpomalení. Podobně by bylo možné nalézt vhodnější rozdělení pomocného pole mezi vlákny v paralelní verzi hybridního algoritmu, než jaké bylo navrženo v této práci.

Při paralelizaci *Block Mergesortu* pak byla zmíněna možnost vytvoření algoritmu, který pracuje zrcadlově vzhledem k průběhu algoritmu *Block Merge*, pro slučování polí, kde pravá část je delší než levá. Místo toho byla pouze provedena malá úprava původního algoritmu, která problém vyřešila, ne však ideálně. Návrh lepšího algoritmu by tedy mohl být předmětem dalšího výzkumu.

Závěr

Cílem této práce byla implementace algoritmů stabilního řazení v jazyce C++, následně jejich optimalizace a paralelizace pomocí technologie OpenMP. Konkrétně pak algoritmů Insertion sort a in-place a out-of-place variant algoritmu Mergesort. Dalším cílem této práce bylo srovnání časové a paměťové náročnosti těchto dvou variant Mergesortu. Následně měl být vytvořen hybridní algoritmus kombinující principy obou variant Mergesortu. Poté změření jeho výkonnosti a porovnání s výkonností vybraných existujících implementací stabilního řazení.

V rámci této práce byly vytvořeny sekvenční a paralelní implementace různých variant in-place a out-of-place Mergesortu. Také byl vytvořen a implementován hybridní algoritmus kombinující principy nejlepších z in-place a out-of-place variant za účelem efektivního využití poskytnutého pomocného pole různých délek. Tento algoritmus byl z hlediska jeho výkonnosti následně porovnán s vybranými existujícími implementacemi stabilního řazení, přičemž v některých případech se ukázal jako lepší řešení, v jiných zase naopak jako horší. Všechny cíle této práce byly tedy splněny.

Přínosem této práce jsou implementace různých algoritmů jak pro sekvenční, tak paralelní stabilní řazení, které se ukázaly jako kompetitivní s existujícími řešeními. Tyto algoritmy by tedy mohly potenciálně nalézt využití i v praxi. Dalším přínosem je autorem vytvořený hybridní algoritmus, který umožňuje v případě jeho sekvenční verze poměrně efektivní využití různých délek pomocného pole. V případě paralelní verze není toto využití natolik efektivní, ale stále je znatelné.

Literatura

[1] Malík, J.; Suchý, O.; Tvrdík, P.; aj.: BI-AG1 – Přednáška č. 9 – QuickSort, dolní odhad složitosti řazení, speciální algoritmy řazení, Listopad 2016, [cit. 2017-04-01]. Dostupné z: https://edux.fit.cvut.cz/archive/B161/BI-AG1/_media/lectures/bi-ag1-p9-handout.pdf

[2] in-place sorting algorithm | planetmath.org. [online], [cit. 2018-01-06]. Dostupné z: http://planetmath.org/inplacesortingalgorithm

[3] Šimeček, I.: Kompilátorové optimalizace I: Modely chování skryté (cache) pamětí, 2016, [cit. 2018-01-07]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-EIA/_media/lectures/komp_cache.pdf

[4] Lórencz, R.; Hlaváč, J.; Zahradnický, T.: Pameťová hierarchie, návrh skryté paměti 1, 2012, [cit. 2018-01-07]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-APS/_media/aps_mem_1.pdf

[5] Insertion sort. [online], [cit. 2018-01-06]. Dostupné z: https://www.algoritmy.net/article/8/Insertion-sort

[6] Merge sort. [online], [cit. 2018-01-06]. Dostupné z: https://www.algoritmy.net/article/13/Merge-sort

[7] Juszczak, C.: Fast mergesort implementation based on half-copying merge algorithm. Duben 2007, [online], [cit. 2018-01-08]. Dostupné z: http://kicia.ift.uni.wroc.pl/algorytmy/mergesortpaper.pdf

[8] Dudziński, K.; Dydek, A.: On a stable minimum storage merging algorithm. *Information Processing Letters*, ročník 12, č. 1, 1981: s. 5–8, ISSN 0020-0190, doi:10.1016/0020-0190(81)90065-X. Dostupné z: http://www.sciencedirect.com/science/article/pii/002001908190065X

[9] Katajainen, J.; Pasanen, T.; Teuhola, J.: Practical In-place Mergesort. *Nordic J. of Computing*, ročník 3, č. 1, Březen 1996: s. 27–40, ISSN 1236-6064.

[10] Kim, P.-S.; Kutzner, A.: Ratio Based Stable In-place Merging. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation*, TAMC'08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-79227-9, 978-3-540-79227-7, s. 246–257.

[11] Pardo, L. T.: Stable Sorting and Merging with Optimal Space and Time Bounds. *SIAM Journal on Computing*, ročník 6, č. 2, 1977: s. 351–372, doi:10.1137/0206025.

[12] OpenMP 4.5 API C/C++ Syntax Reference Guide. 2015, [online], [cit. 2018-01-05]. Dostupné z: `http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf`

[13] Block swap algorithm for array rotation - GeeksforGeeks. [online], [cit. 2018-01-08]. Dostupné z: `https://www.geeksforgeeks.org/block-swap-algorithm-for-array-rotation/`

[14] Geffert, V.; Katajainen, J.; Pasanen, T.: Asymptotically efficient in-place merging. *Theoretical Computer Science*, ročník 237, č. 1, 2000: s. 159–181, ISSN 0304-3975, doi:10.1016/S0304-3975(98)00162-5. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0304397598001625`

[15] Hwang, F. K.; Lin, S.: A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput*, 1972, doi:10.1137/0201004.

[16] Langr, D.; Šimeček, I.; Tvrdík, P.: Parallel sorting in OpenMP, 2017, [cit. 2018-01-08]. Dostupné z: `https://edux.fit.cvut.cz/courses/MI-PDP.16/_media/en/lectures/mie-pdplecture06-openmpsorting.pdf`

[17] Robison, A. D.: A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP | Intel® Software. Duben 2014, [online], [cit. 2018-01-05]. Dostupné z: `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`

[18] McFadden, M.: GitHub - BonzaiThePenguin/WikiSort: Fast and stable sort algorithm that uses O(1) memory. Public domain. 2014, [online], [cit. 2018-01-05]. Dostupné z: `https://github.com/BonzaiThePenguin/WikiSort`

Příloha **A**

Seznam použitých zkratek

GNU GNU is Not Unix

RAM Random Access Memory

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_Čermák_Michal_2018.pdf` ..... text práce ve formátu PDF