FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Efektivní paralelní vícecestný Quicksort algoritmus
Student:	Bc. Ondřej Voronecký
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové systémy a sítě
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1) Seznamte se s problematikou paralelního řazení dat v prostředí vícevláknových aplikací se sdílenou pamětí. Zaměřte se zejména na sekvenční implementaci vícecestného Quicksort algoritmu. [1]
2) Prostudujte paralelní implementaci Quicksort algoritmu. [2]
3) Analyzujte možnosti kombinace Quicksort algoritmu s dalšími (insertion sort, heapsort, atd.).
4) Na základě předchozích bodů navrhněte a implementujte paralelní verzi vícecestného Quicksort algoritmu pomocí OpenMP. [1]
5) Porovnejte výslednou implementaci s existujícími implementacemi paralelních algoritmů dostupných pro C++.[4]

[1] Wild, Sebastian: Dual-pivot and beyond: The potential of multiway partitioning in quicksort. it - Information Technology. 60. 173-177. 10.1515/itit-2018-0012, 2018
[2] AQsort: Scalable Multi-Array In-Place Sorting with OpenMP https://www.scpe.org/index.php/scpe/article/view/1207
[3] Prof. P. Tvrdík, přednášky NI-PDP, ČVUT FIT
[4] Bc. Klára Schovánková: Paralelní řazení v C++ 11, DP ČVUT FIT 2019

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 20. dubna 2022 v Praze.

Diplomová práce

EFEKTIVNÍ PARALELNÍ VÍCECESTNÝ QUICKSORT ALGORITMUS

Bc. Ondřej Voronecký

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
3. ledna 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych touto cestou poděkoval vedoucímu této práce doc. Ing. Ivanu Šimečkovi, Ph.D. za ochotu a cenné rady, které vedly ke zkvalitnění této práce. Také bych chtěl poděkovat své rodině za poskytnutou podporu a zázemí při psaní.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, 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, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 3. ledna 2023 .....

Abstrakt

V této diplomové práci je představena nová verze paralelního in-place Quicksort algoritmu MPQsort pro řazení polí, s využitím OpenMP pro paralelizaci. Dosavadní implementace využívají pro rozdělování pouze jednoho pivota. MPQsort implementuje paralelní vícecestné rozdělování a jedná se o první algoritmus svého druhu. V práci jsou diskutována sekvenční vícecestná rozdělování, následně paralelní dvoucestná rozdělování a na jejich základě navržena a implementována paralelní vícecestná varianta. Poté je provedeno experimentální vyhodnocení efektivity algoritmu a porovnání s existujícími implementacemi. V experimentech MPQsort dosahuje dobrých výsledků a z uvažovaných algoritmů se umístil na druhém místě v oblasti řazení náhodných čísel. Naopak v případě jiných typů uspořádání dat dosahuje i nejlepších výsledků.

Klíčová slova quicksort, řazení, paralelní, efektivní, vícecestný, vícecestné rozdělení, openmp, c++, algoritmus, pivot, MPQsort

Abstract

A new version of the parallel in-place Quicksort algorithm MPQsort for array sorting is presented in this thesis, using OpenMP for parallelization. Current implementations use only one pivot for element partitioning. On the other hand, MPQsort implements parallel multi-way partitioning and so is the first algorithm of its kind. Sequential multi-way partitionings are discussed in the first part of the thesis, followed by parallel two-way partitioning. Based on the gathered information is designed and implemented parallel multi-way partitioning. Implementation was followed by an experimental evaluation of its efficiency and comparison with other implementations. MPQsort achieves good results in experiments and among the other considered algorithms ranked second in terms of sorting randomly generated numbers. Conversely, it sometimes achieves the best results for other types of data arrangements.

Keywords quicksort, sorting, parallel, effective, multi-way, multi-way partitioning, openmp, c++, algorithm, pivot, multipivot, MPQsort

Seznam zkratek

API	Application Programming Interface
BQS	Balanced Quicksort
ET	elements_table
ETi	elements_table_insertions
ETI	elements_table_insertions
ETIi	elements_table_insertions_index
F&A	fetch-and-add
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
MPQsort	Multi-way Parallel Quicksort Algorithm
OpenMP	Open Multi-Processing
QS	Quicksort
RAM	Random Access Machine
STL	Standard Template Library
YBB	Yaroslavskiy-Bentley-Bloch

Úvod

Problém řazení je jedním z nejběžnějších a nejčastějších algoritmických úloh, se kterými se můžeme setkat. Můžeme na něj narazit prakticky kdekoliv, ať už při analýze dat, hledání mediánu nebo jako dílčí krok jiných algoritmů. Z této skutečnosti pramení silná motivace pro jeho maximální efektivitu, jelikož je součástí tak velkého množství algoritmických úloh a zároveň se zvyšují nároky na velikost řazených dat. Maximální efektivity chceme docílit z hlediska časových i paměťových nároků na počítačový systém.

Algoritmů řešících problém řazení existuje nepřeberné množství, přičemž každý nabízí odlišné vlastnosti a dává si různé cíle. Například algoritmus *Mergesort* garantuje vždy stejnou časovou složitost (je datově necitlivý) na úkor nízké paměťové efektivity (out-of-place). *Quicksort* je na druhou stranu datově citlivý, ale zpravidla implementovaný jako in-place a tedy paměťově efektivní. V průměru však *Quicksort* dosahuje stejné časové složitosti jako *Mergesort*. Z toho důvodu je nejvíce implementovaným algoritmem a součástí mnoha standardních knihoven programovacích jazyků.

S příchodem vícejádrových procesorů dává smysl tyto efektivní sekvenční algoritmy implementovat jako paralelní, tím prostředky procesoru lépe využít a algoritmus zrychlit. Paralelní verze *Quicksort* algoritmu je součástí knihovny *STL* jazyka C++ od standardu verze 17, ale ne všechny překladače jej podporují. Běžné implementace paralelního Quicksort algoritmu však využívají jednoho pivota, což je nejrozšířenější způsob implementace. V poslední době se však nabízí otázka, zda by se využitím vícero pivotů a implementací tzv. vícecestného paralelního Quicksort algoritmu nedosáhlo vyšší časově efektivity. Důvod možného zrychlení pomocí vícero pivotů je prohlubující se rozdíl v rychlosti procesoru a hlavní paměti. Rychlost procesoru roste ročně více, nežli rychlost pamětí a v dnešní době rozdíl dosáhl meze, kdy by se vyplatilo snížit počet přístupů do paměti na úkor počtu operací procesoru.

Tato práce se zaměřuje na analýzu vícecestného *Quicksort* algoritmu, prozkoumání dosavadních variant a jeho implementací. Na základě této analýzy bude navržen nový vícecestný Quicksort algoritmus v prostředí vícevláknových aplikací se sdílenou pamětí. Pro paralelizaci bude využita externí knihovna OpenMP, která se řadí k nejvyužívanějším v oblasti paralelního programování. Následovat bude rozsáhle měření a vyhodnocení efektivity navrženého algoritmu a porovnání s dosavadními paralelními implementacemi pro jazyk C++.

V kapitole 1 se seznámíme se základními řadícími algoritmy. Součástí bude uvedení jejich časové, paměťové složitosti a následně i dodatečných vlastností, které mohou mít nezanedbatelný vliv při návrhu a implementaci našeho algoritmu.

Po zadefinování potřebných pojmů a uvedení do problematiky řazení polí následuje kapitola 2 pojednávající o sekvenčních variantách algoritmu Quicksort. Probereme základní dvoucestná rozdělování, která sloužila jako podklad pro implementaci vícecestné varianty. Díky tomu získáme představu, jak vícecestné rozdělování funguje, a to nám pomůže při implementaci paralelního vícecestného rozdělování. Následuje empirické vyhodnocení jejich vlastností a výběr

implementace, která bude využita i v rámci paralelní verze Quicksortu.

Na začátku kapitoly 3 uvedeme dosavadní paralelní implementace algoritmu. Díky tomu získáme vhled do problematiky a zjistíme nutné požadavky na efektivní implementaci. V tuto chvíli máme všechny potřebné informace pro analýzu a návrh nového vícecestného paralelního Quicksort algoritmu s názvem MPQsort. Autor této práce si není vědom jiné implementace či publikace, která by pojednávala o paralelním vícecestném rozdělování, a z toho důvodu se jedná o první implementaci svého druhu.

Součástí implementace algoritmu je příprava testovacího prostředí, otestování korektnosti algoritmu a příprava testovacích dat pro vyhodnocení jeho výsledné efektivity. To vše je probráno v kapitole 4.

Závěrečná kapitola 5 celé práce diskutuje nastavení parametrů algoritmu, vyhodnocení rychlosti řazení a porovnání s existujícími implementacemi. Srovnání algoritmů bylo provedeno na rozsáhlých polích, různých datových typech a uspořádáních.

Kapitola 1
Řadící algoritmy

Řazení dat patří mezi nejzákladnější problémy v informačních technologiích a algoritmizaci. Zpravidla se nutnost seřazení dat vyskytuje jako podproblém nějakého jiného algoritmu, a proto je potřeba, aby byly řadící algoritmy co nejvíce efektivní. Seřazení dat může urychlit následný běh programu či algoritmu. Pro problém řazení existuje několik různých algoritmů splňujících odlišné vlastnosti, které budou probrány v kapitole 1.2. Ač se jedná o elementární a jeden z nejstarších problémů, považuje se stále za aktuální a snaha o jeho efektivnější implementaci neustává. Jednou ze snah je efektivní implementace paralelní verze řadícího algoritmu, jelikož jsou takřka všechny procesory vícejádrové a taková implementace může přinést podstatné zrychlení vůči sekvenční verzi.

Problém řazení spočívá v převedení vstupní posloupnosti n prvků $A = (a_1, a_2, a_3, \ldots, a_n)$ na výstupní permutací prvků $B = (b_1, b_2, b_3, \ldots, b_n)$ takovou, že pro všechny prvky posloupnosti B bude platit $b_i \leq b_{i+1}$ pro všechna i jdoucí od 0 do $n - 1$. Seřazenost výstupních dat závisí na využitém operátoru. Pro jednoduchost budeme v této práci pracovat pouze s operátorem „$<$“ (menší než), ale data mohou být seřazena i v opačném pořadí. [1] [2] Pod termínem posloupnost si také můžeme představit pole, které je vstupem řadících algoritmů v programovacích jazycích.

V této kapitole probereme základní vlastnosti řadících algoritmů, dle kterých se dají kategorizovat. Díky tomu jsme schopni vybrat ten nejvhodnější na základě povahy dat určených k seřazení a aktuálních potřeb. Obecný řadící algoritmus, který by byl nejlepší pro všechny typy dat a potřeby využití, totiž neexistuje. Dále probereme základní řadící algoritmy, které se stále široce využívají a zaměříme se na ty, které nám pomohou v implementaci paralelního vícecestného Quicksort algoritmu. Uvedeme si možné implementace klasické dvoucestné verze Quicksortu (s využitím jednoho pivota) a na závěr probereme verze, které implementují vícecestné rozdělování využívající dva a více pivotů. Z těchto variant Quicksort algoritmu vybereme tu nejvhodnější, kterou budeme v paralelní verzi využívat pro řazení krátkých polí.

1.1 Vlastnosti řadících algoritmů

Řadící algoritmy se dají kategorizovat na základně jejich vlastností. Tyto vlastnosti se neomezují jen na časovou a paměťovou složitost, ale i na stabilitu dat, rekurzivitu a jejich citlivost na vstupní data. [2] Každý řadící algoritmus v určitých vlastnostech vyniká, ale zpravidla je tomu tak na úkor jiných. Algoritmus, který by ve všech vynikal, neexistuje. Například datová citlivost může v určitých případech zcela změnit časovou složitost (především u Quicksort algoritmu) a jinak efektivní algoritmus převést na neefektivní. V naší implementaci budeme Quicksort kombinovat s jinými řadícími algoritmy, abychom jeho některé negativní vlastnosti byli schopni potlačit a dosáhnout vyšší efektivity.

Časová složitost

Časová složitost je jednou z nejdůležitějších vlastností algoritmu. Určuje asymptotický vztah doby běhu vůči velikosti zpracovávaných dat. Velikostí rozumíme délku vstupu (počet elementů určených k seřazení) a budeme ji značit pomocí n. Časová složitost se udává jako tzv. asymptotická složitost, která určuje složitost pro veliká n a díky této skutečnosti se mohou zanedbat multiplikativní a aditivní konstanty. Určují se zpravidla tři typy mezí časové složitosti. Dolní mez časové složitosti nám říká, že algoritmus nemůže běžet rychleji a býti efektivnější než stanovená mez (značí se Ω). Průměrná složitost nám naopak říká dobu běhu algoritmu obecně, napříč různými typy vstupů. Může se totiž stát, že je algoritmus obecně velmi rychlý a existuje pouze velmi malé množství vstupů, pro které dosahuje vyšší časové složitosti. V takovém případě je lepší počítat s jeho průměrnou složitostí, která je přesnější. Průměrná složitost je dána jako průměr doby běhu přes všechny možné vstupy o délce n. [2] Posledním typem je tzv. horní mez časové složitosti, vyjadřující dobu běhu algoritmu v nejhorším případě (pro nejhorší možný vstup pro daný algoritmus). Ta je v případě popisu vlastností algoritmu uváděna nejčastěji a značí se pomocí řeckého písmene O. [3]

Běžné časové složitosti řadících algoritmů, se kterými se můžeme setkat, jsou $O(n^2)$ nebo $O(n \log n)$. Složitost $\Omega(n \log n)$ je i dolní mez řadících algoritmů, pokud budeme uvažovat porovnávací model RAM. [4] Jedná se o teoretický výpočetní model popisující elementární operace nad paměťovými buňkami. Buňky jsou celočíselně adresovány, přičemž každá z nich je schopna uložit právě jedno číslo. Operacemi rozumíme základní aritmetické, logické operace a řídící instrukce. [2] Je dokázáno, že pokud budeme uvažovat algoritmus, který má být schopen seřadit jakoukoli vstupní posloupnost, neklást žádné nároky na její vlastnosti a pracovat v RAM modelu, pak není schopen učinit rychleji než výše uvedená dolní mez. Existují algoritmy jako *Counting sort*, schopné seřadit posloupnosti rychleji než v čase $\Omega(n \log n)$, ale mají poměrně striktní požadavky na vlastnosti vstupu (zpravidla velmi nízká mohutnost množiny oboru hodnot, kterých jednotlivé elementy nabývají).

Datová citlivost

Datová citlivost řadícího algoritmu určuje vliv vstupní posloupnosti na časovou složitost algoritmu. Pokud je algoritmus datově necitlivý, pak to znamená, že pro všechny délky n vstupní posloupnosti A bude mít identickou časovou složitost. Z toho vyplývá, že jsou si hodnoty všech mezí rovny, tedy $\Omega = O$. Pokud je datově citlivý, může vstup ovlivnit dobu běhu a počet operací nutných k dosažení výsledku. Quicksort je příkladem datově citlivého algoritmu, jelikož uspořádání dat může v určitých případech ovlivnit výběr pivota a mít za následek nevyváženost velikosti vzniklých segmentů a tím zmenšení/zvětšení celkového počtu operací. [2] [4] Jako zástupce datově necitlivého algoritmu můžeme uvést *Mergesort* nebo *Heapsort*, které řadí vždy v čase $\Theta(n \log n)$.

Citlivost na vstupní data může být, přínosem pokud víme, že je posloupnost skoro seřazena a zbývá seřadit pouze malé množství prvků. V tom případě je výhodné využít např. *Shaker sort* s kontrolou seřazenosti posloupnosti, který by byl prakticky rychlejší než datově necitlivý algoritmus.

Paměťová složitost

Paměťová složitost (někdy také prostorová složitost) algoritmu uvádí jeho nároky na operační paměť po dobu jeho výpočtu. Stejně jako v případě časové složitosti se uvádí asymptoticky a využívá identické notace pro různé meze. Zpravidla se však zajímáme čistě o nejvyšší možnou prostorovou složitost (značenou O), jelikož systém má paměti omezené množství a při běhu algoritmu ji nesmíme přesáhnout. Často platí, že se časová složitost dá snížit na úkor prostorové

a při výběru algoritmu se v případě systému s dostatečným množstvím hlavní paměti přikláníme spíše k lepší časové než paměťové složitosti.

Prostorová složitost se uvádí ve vztahu k velikosti vstupu n. Vždy uvažujeme pouze dodatečnou paměť, potřebnou při výpočtu. Množství paměti, kterou zabírají vstupní data pří výpočtu neuvažujeme. Pokud potřebná paměť algoritmu není závislá na velikosti vstupu, pak ji značíme jako konstantní $O(1)$ (pro výpočet je vyžadováno pouze konstantní množství pomocných proměnných). [2] Na základě této skutečnosti rozlišujeme dva typy algoritmů (v případě řazení obzvláště důležitá vlastnost) a to *in-place* a *out-of-place*.

In-place algoritmy nepožadují dodatečnou paměť závislou na velikosti vstupu a pro svůj běh požadují pouze konstantní množství dodatečné paměti (lokální proměnné). Jejich paměťová složitost je tedy $O(1)$. Takové algoritmy jsou obzvláště výhodné, pokud máme paměti nedostatek a algoritmus zpracovává veliké množství dat, které by se nemuselo do paměti vejít. Zástupcem z řad řadících algoritmu je Insertion sort nebo Shaker sort.

Out-of-place algoritmy pro svůj běh vyžadují alokaci dodatečné paměti a ta je závislá na velikosti zpracovávaného vstupu. Nejznámějším zástupcem z řadících algoritmů je Mergesort, který má prostorovou složitost $O(n)$. Vedle vstupního pole si totiž potřebuje alokovat další, které bude mít identickou velikost n. Existují i implementace, které jsou schopny pracovat in-place, ale je tomu tak na úkor časové složitosti.

Stabilita řazení

V určitých případech chceme data seřadit dle velikosti, ale chceme přitom zachovat jejich relativní pořadí v případě, kdy jsou si elementy rovny. Přesně to vystihuje pojem stabilita řadícího algoritmu. Pokud se tedy ve vstupní posloupnosti vyskytují prvky, které jsou si rovny na základě poskytnutého operátoru, pak je řadící algoritmus stabilní právě tehdy, když se jejich vzájemné pořadí na výstupu shoduje s jejich pořadím na vstupu. Formálněji pokud $a_i = a_j$ pro $i < j$, pak je a_i ve výstupu před a_j. [2]

Zástupci stabilních řadících algoritmů jsou například Insertion sort nebo Bubble sort.

Rekurzivita

Rekurzivita řadícího algoritmu není často uváděna, ale může být poměrně důležitou vlastností, jelikož může ovlivňovat výslednou implementaci řadícího algoritmu a praktickou dobu běhu. Rekurzivní algoritmy dávají smysl u vstupů větších velikostí, ale přinášejí nezanedbatelné zpomalení oproti nerekurzivním v případě malých vstupů (přesnou hranici je třeba určit empiricky). Přílišné volání funkce pro malé vstupy může způsobit, že její samotné volání, uložení lokálních proměnných na zásobník a jejich následné čtení bude trvat déle, než samotné zpracování dat. Opakovanému volání funkce se dá zamezit pomocí simulace zásobníku datovou strukturou *stack*. To však není vždy vhodné, jelikož tím může být implementace méně přehledná a složitější. Pokud na řešení daného problému existuje nerekurzivní algoritmus, je vhodné jej využít pro řešení dostatečně malých podproblémů vzniklých v rámci rekurzivního algoritmu. Tím jsme schopni vzniklou neefektivitu eliminovat.

Nejznámějšími zástupci rekurzivních algoritmů jsou Quicksort a Mergesort. Z řad nerekurzivních pak Insertion sort a Bubble sort.

1.2 Základní řadící algoritmy

Základní řadící algoritmy jsou zpravidla jednoduché na implementaci, jsou krátké, in-place, ale mají kvadratickou časovou složitost. [2] Mezi nejznámější patří Bubble sort, Selection sort a Insertion sort. Tyto algoritmy jsou nerekurzivní a in-place, proto je jejich využití vhodné pro řazení krátkých posloupností a nebo jako součást rekurzivního řadícího algoritmu (důvody uvedeny

v předcházejících odstavcích). Z praktického hlediska je z výše uvedených nejlepší Insertion sort. V praxi totiž dosahuje vyšší rychlosti a proto jej využijeme jako jeden z pomocných sekvenčních nerekurzivních algoritmů v rámci naší implementace sekvenčního vícecestného Quicksort algoritmu. [5]

O něco složitějším in-place algoritmem je Heapsort. Jeho výhodou je jeho časová složitost, která je vždy $O(n \log n)$. Bude proto druhým algoritmem využitým při implementaci. Důvody pro jeho výběr jsou ve větších detailech probrány v sekci 2.4.

Níže je uvedena tabulka s přehledem známých řadících algoritmů a jejich vlastností. Časová složitost je uváděna v průměrném případě (značeno symbolem Θ). [2] Ve sloupcích, kde je uvedeno Ano/Ne, záleží na konkrétní implementaci, zda bude mít algoritmus danou vlastnost. U algoritmu Quicksort je uvedena logaritmická paměťová složitost, a to i v případě, kdy je implementován jako in-place, což může být poněkud matoucí. Je tomu proto, že z definice nemůže být konstantní, pokud je závislá na vstupu. Při každém rekurzivním volání je třeba uložit lokální proměnné (pivot a jiné proměnné potřebné k výpočtu) na zásobník. Množství paměti je zanedbatelné, ale kvůli korektnosti bylo třeba tuto skutečnost uvést.

Algoritmus	**Časová slož.**	**Paměťová slož.**	**Dat. cit.**	**Stabilita**	**Rekurzivita**
Bubble sort	$\Theta(n^2)$	$\Theta(1)$	Ano/Ne	Ano	Ne
Insertion sort	$\Theta(n^2)$	$\Theta(1)$	Ano	Ano	Ne
Mergesort	$\Theta(n \log n)$	$\Theta(n)$	Ne	Ano	Ano/Ne
Heapsort	$\Theta(n \log n)$	$\Theta(1)$	Ne	Ne	Ne
Quicksort	$\Theta(n \log n)$	$\Theta(\log n)$	Ano	Ne	Ano

Tabulka 1.1 Vlastnosti základních řadících algoritmů.

Kapitola 2

Sekvenční Quicksort

Quicksort se dá považovat za nejpoužívanější řadící algoritmus, za což vděčí jeho dobrým vlastnostem a faktu, že je in-place. V praxi tento algoritmus dosahuje vyšší rychlosti než ostatní algoritmy, pokud je dobře implementován a optimalizován. Proto je široce využíván ve standardních knihovnách nejznámějších jazyků jako je C++ či Java. Z toho důvodu je snahou Quicksort co nejvíce optimalizovat, a to pomocí kombinací s jinými algoritmy, výběrem pivota či implementací využívající více pivotů (více v sekci 2.2).

V této sekci si probereme strukturu algoritmu, jeho dílčí části a zadefinujeme potřebnou terminologii využívanou v celém zbytku práce.

2.1 Struktura a popis algoritmu

Quicksort pracuje v RAM modelu a proto pro něj platí stejné limitace z hlediska časové složitosti, jako u jiných základních řadících algoritmů (dolní mez $\Omega(n \log n)$). Hlavní myšlenkou klasického Quicksortu je vybrat prvek ze vstupní posloupnosti, kterému říkáme *pivot*, a přerozdělit vstupní posloupnost do dvou segmentů. Z toho důvodu se řadí mezi klasické algoritmy *rozděl-a-panuj*. Vstupní problém rozdělíme na několik menších podproblémů a ty řešíme obdobným způsobem rekurzivně. První segment bude obsahovat prvky menší než pivot a druhý segment bude obsahovat prvky větší nebo rovny pivotu, jako je vidět na obrázku 2.1. Procesu přemístění prvků do správných paměťových buněk za účelem splnění tohoto uspořádání se říká *rozdělování*. V tomto případě se jedná o tzv. *dvoucestné* rozdělování, protože byla posloupnost rozdělena na dvě části a každá část seskupuje prvky na základě jejich vztahu vůči pivotovi. Některé publikace uvádějí termín třícestné rozdělování i v případě využití jednoho pivota. Posloupnost se rozdělí na tři segmenty, přičemž prostřední z nich bude obsahovat prvky rovny pivotu, což je zachyceno na obrázku 2.2. Z našeho pohledu se stále jedná o rozdělování dvoucestné, jelikož prvky rovny pivotu jsou již na správném místě a proto mohou být zcela vynechány z následujících výpočtů. Tedy i v tomto případě jsme problém rozdělili na dva podproblémy a ne na tři. Některé publikace definují uvedené schéma jako rozdělování pomocí tzv. *širokého pivota*. [2] [4] [6]

Poté co je pole rozděleno na segmenty, je na každý segment zvlášť zavolán Quicksort rekurzivně a proces se opakuje až do splnění ukončovací podmínky. Poté co jsou seřazeny všechny segmenty, vznikne seřazené výstupní pole.

$< P$	P	$\geq P$

Obrázek 2.1 Umístění prvků pole po zpracování dvoucestným rozdělováním.

$< P$	P	$= P$	$> P$

Obrázek 2.2 Umístění prvků pole po zpracování se segmentem pro prvky rovny pivotu.

Tabulka 1.1 uvádí průměrnou časovou složitost Quicksortu jako $\Theta(n \log n)$, která je však v nejhorším případě až kvadratická $O(n^2)$. Degradaci na kvadratickou složitost je možno potlačit výběrem vhodného pivota takovým způsobem, aby rozděloval pole na segmenty o přibližně stejné velikosti a kombinací s jinými řadícími algoritmy, popsanými v sekci 2.4.

Quicksort je v základních a běžných implementacích nestabilním řadícím algoritmem. Existují implementace dodržující stabilitu řazení, ale ty vyžadují dodatečnou paměť během výpočtu. [2] Také je považován za in-place algoritmus i navzdory nutné alokaci $O(\log n)$, což bylo popsáno v 1.2. V neposlední řadě je datově citlivý, jelikož vstupní data mají vliv na kvalitu výběru pivota a tedy i velikost segmentů po rozdělování.

Pseudokód 2.1 zachycuje triviální implementaci řídící části algoritmu v případě jednoho (dvoucestný) a dvou pivotů (třícestný). V případě vícecestných variant by byla řídící část implementována obdobně. Můžeme si povšimnout koncové rekurze, která se dá přepsat na cyklus a tím ušetřit jedno zbytečné rekurzivní volání. V rámci řídící části se v praxi implementuje volání jiných řadících algoritmů na základě aktuální velikosti vstupu, hloubky rekurze a jiných zvolených parametrů.

Výpis kódu 2.1 Pseudokód triviální implementace řídící části algoritmu Quicksort.

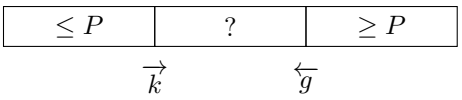
```
1  // Základní dvoucestná varianta
2  Quicksort(A, lp, rp) {
3      if (lp < rp) {
4          pi = Partition(A, lp, rp)
5          Quicksort(A, lp, pi - 1)
6          Quicksort(A, pi + 1, rp)
7      }
8  }
9
10 // Třícestná varianta
11 Quicksort(A, lp, rp) {
12     if (lp < rp) {
13         [pi1, pi2] = Partition(A, lp, rp)
14         Quicksort(A, lp, pi1 - 1)
15         Quicksort(A, pi1 + 1, pi2 - 1)
16         Quicksort(A, pi2 + 1, rp)
17     }
18 }
19
20 // Dvoucestná varianta bez koncové rekurze
21 Quicksort(A, lp, rp) {
22     while (lp < rp) {
23         pi = Partition(A, lp, rp)
24         Quicksort(A, lp, pi - 1)
25         lp += pi + 1;
26     }
27 }
28
```

Rozdělování

Rozdělování posloupnosti na segmenty je jádrem algoritmu Quicksort. Jeho kvalita a efektivita určuje celkovou dobu běhu a vlastnosti algoritmu. V této sekci se stručně seznámíme se základními variantami dvoucestného rozdělování, což jsou *Sedgewick-Hoare* a *Lomuto*. [6] Pro následující stránky definujme pivoty jako $P_1 \leq \ldots \leq P_{s-1}$, kde s vyjadřuje počet vzniklých segmentů. Díky tomu jsme schopni při p pivotech rozdělit posloupnost na $s = p + 1$ segmentů. Každý pivot P_i je rozdělovačem segmentů s_i a s_{i+1}. Segmenty se nepřekrývají a díky uspořádání pivotů (pořadí závisí na zvoleném operátoru) tvoří pivoti uspořádanou posloupnost. Po dokončení rozdělování jsou tedy i segmenty vůči sobě správně uspořádány a máme zaručeno, že prvky v daném segmentu zůstanou (není potřeba přesun prvků mezi segmenty) a pouze se může změnit umístění prvků v rámci daného segmentu.

Na následujících řádcích si probereme dva elementární algoritmy rozdělování *Sedgewick-Hoare* a *Lomuto* využívající jednoho pivota, které rozdělují vstupní data do dvou segmentů. Pro zjednodušení značení nechť $P = P_1$ pro oba algoritmy. Algoritmy řeší identický problém, ale přistupují k němu dvěma různými způsoby, přičemž každý má pozitiva i negativa. Jejich pochopení je klíčem k vícecestnému rozdělování. To totiž zpravidla využívá kombinaci obou přístupů a tvoří základ jejich logiky.

Sedgewick-Hoare je starší variantou, která pracuje s dvěma indexy k a g, které se postupným zpracováváním prvků posloupnosti přibližují k sobě zleva a zprava. [6] [7] Index k resp. g se inkrementuje resp. dekrementuje, pokud je daný element ve správném segmentu. Pokud se pod indexy nachází elementy, které spadají do opačného segmentu, pak se elementy prohodí a pokračuje se s dalšími. Indexy k a g se potkají na hranici segmentů, čímž rozdělí data na dvě části. Po vložení pivota na tuto hranici je na své koncové pozici a nově vytvořené segmenty se rekurzivně zpracují obdobným způsobem. Pro lepší představu obrázek 2.3 zachycuje invariant rozdělování posloupnosti. [6] Oblast označena symbolem „?“ bude po skončení rozdělování pokryta levým, resp. pravým segmentem.



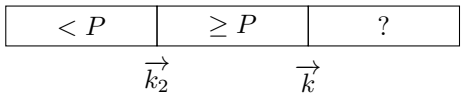
Obrázek 2.3 Invariant *Sedgewick-Hoare* rozdělování.

Pseudokód 2.2 přehledně ukazuje, jak funguje rozdělování *Sedgewick-Hoare*. Funkce *GetPivot* z posloupnosti čísel vybere pivota a vrátí nám jeho index. Výběr pivota funkcí *GetPivot* může být náhodný, medián tří prvků, poslední element, či jiný vhodný způsob. Oba dva segmenty po ukončení rozdělování mohou obsahovat prvky rovny pivotu. Díky tomu se dá předejít degradaci časové složitosti v případě velkého množství identických prvků, jelikož se nepřesouvají do jednoho segmentu a zůstávají na místě, kde jsou. Tento typ rozdělování se standardně využívá u paralelních implementací rozdělování s jedním pivotem, jelikož se dá efektivně a škálovatelně paralelizovat. [7]

Výpis kódu 2.2 Pseudokód *Sedgewick-Hoare* rozdělování.

```
PartitionSedgewickHoare(A, lp, rp)
    pi = GetPivot(A, lp, rp)
    pivot = A[pi]
    k = lp - 1, g = rp + 1
    while (true)
        do k += 1 while (A[k] < pivot)
        do g -= 1 while (pivot < A[g])
        if (k >= g) return g
        swap(A[k], A[g])
```

Lomuto rozdělování má oproti předchozímu jeden zásadní rozdíl. Indexy nejdou proti sobě tak dlouho, dokud se nepotkají, ale oba postupují zleva doprava. Konec rozdělování nastane v tu chvíli, kdy je vedoucí index k větší nebo roven aktuální délce zpracovávaného vstupu n. Indexy jdoucí zleva postupně označíme k_2 a k, přičemž k_2 je rozdělovačem dvou vznikajících segmentů. Po získání pivota jej umístíme jako poslední prvek vstupu a na konci rozdělování jej přesuneme na správné místo. Obrázek 2.4 demonstruje schéma průběhu algoritmu. Část označená symbolem „?“ ze začátku pokrývá celou vstupní posloupnost.



Obrázek 2.4 Invariant *Lomuto* rozdělování.

Tento přístup je jednodušší z hlediska implementace a je zpravidla uváděn jako příklad implementace Quicksortu. Na druhou stranu je v praxi méně efektivní a v případě velkého množství identických prvků dochází k degradaci na $O(n^2)$, pokud nejsou zavedeny optimalizační techniky. Dále není vůbec vhodný z hlediska paralelizace a nenabízí jednoduchý způsob, jak dílčí části vstupu přerozdělit mezi vlákna. [1] Pseudokód 2.3 ukazuje možnou implementaci algoritmu.

Výpis kódu 2.3 Pseudokód *Lomuto* rozdělování.

```
PartitionLomuto(A, lp, rp)
    pi = GetPivot(A, lp, rp)
    pivot = A[pi]
    swap(A[pi], A[rp])
    k2 = lp
    for (k = lp; k < rp; k += 1)
        if (A[k] < pivot)
            swap(A[k2], A[k])
            k2 += 1
    swap(A[k2], A[rp])
    return k2
```

Správné pochopení algoritmů 2.3 a 2.2 je stěžejním pro kapitolu vícecestného rozdělování. V implementacích vícecestných rozdělování se využívají obě varianty různými způsoby v závislosti na zvoleném počtu pivotů. Výsledná implementace je poté složitější.

Výběr pivota

Výběr pivota je obzvláště důležitý krok pro efektivitu Quicksortu. Vlastnosti pivota určují kvalitu rozdělování, od kterého chceme, aby po jeho dokončení byly výsledné segmenty velikostně co možná nejpodobnější. V takovém případě využíváme maximální potenciál algoritmu, předcházíme jeho degradaci na kvadratickou časovou složitost a maximálně se přiblížíme k průměrné časové složitosti.

V případě dvoucestné varianty chceme posloupnost rozdělit na dvě velikostně podobné části, čímž problém hledání pivota můžeme definovat jako problém hledání mediánu (nebo elementu blízko mediánu).

Triviální implementace volí jako pivota první, resp. poslední prvek posloupnosti. Takové řešení může docílit průměrné časové složitosti v případě, kdy víme, že je posloupnost tvořena náhodnými prvky z uniformního rozdělení a není seřazena. Díky tomu je takový výběr vlastně náhodný prvek z posloupnosti a to, že se trefíme do "skoromediánu", má pravděpodobnost $\frac{1}{2}$. To však klade velmi striktní požadavky na data a v praxi by byl takový algoritmus nepoužitelný. V případě seřazené posloupnosti by takový výběr znamenal volení nejmenšího, resp. největšího prvku, což by mělo za důsledek degradaci na kvadratickou časovou složitost.

Dalším přístupem je výběr pivota jako náhodný prvek z posloupnosti. Tím se eliminuje nejhorší případ pro předem seřazená data a v průměru bychom se měli více přiblížit průměrné složitosti. Problém výběru jsme tímto redukovali, ale kvalita výběru pivota stále nebude optimální. Náhodně vybraný prvek nemusí být dostatečně blízko mediánu a pravděpodobnost výběru skoromediánu je stále nízká v případě dat z uniformního rozdělení. Z toho důvodu se využívají následující přístupy k výběru pivota, které se využívají v praxi.

Medián-tří vybere tři prvky z posloupnosti a z nich vybere medián. Výběr prvků může být náhodný a nebo fixní (element první, poslední a prostřední). Tato varianta podstatně zvýší pravděpodobnost výběru pivota blízko mediánu oproti dříve uvedeným přístupům a je pro reálná data běžně užívána. Náhodný výběr s sebou přináší dobu strávenou generováním náhodných indexů posloupnosti a proto se využívají fixní indexy, které se v praxi osvědčily. Dalším vylepšením je varianta *Medián-pěti*, která obdobně vybere místo tří prvků pět a z nich vybere medián. Třetí známou technikou je výběr pivota jako *medián-tří-mediánů-tří-prvků*. Nejrychlejším způsobem je dle [8] technika *Medián-pěti*.

Nevýhodou je čas strávený na výběru prvků a generování náhodných indexů v poli pro kratší posloupnosti (pokud se rozhodneme pro náhodný výběr indexů). Naopak u delších posloupností je výběr mediánu pomocí metod výše nedostatečný a přesnost výběru se může u velmi rozsáhlých posloupností snížit. Proto je vhodnější možností vybrat l elementů (tzv. *sampling*), ty seřadit a z nich vybrat medián. [9] Hodnota l bude vždy relativní vůči aktuální fázi algoritmu. V případě začátku Quicksort algoritmu na přesnosti výběru pivota záleží více, jelikož se zpracovává největší množství dat a chceme docílit co nejpodobnějších velikostí jednotlivých segmentů. Naopak ve větší hloubce rekurzivního stromu volání, kdy řadíme relativně o dost menší posloupnosti, by kvalitní výběr mohl naopak přinést zpomalení. Tímto způsobem docílíme přesnějšího výběru pivota a eliminujeme časovou náročnost pro malé velikosti vstupu. [9] Varianta využívající sampling je také vhodná pro výběr více pivotů, kde bude pivot P_i určen jako element z posloupnosti l vybraných prvků, indexovaný jako $l\frac{i}{p+1}$, kde p je počet pivotů a i index pivota. Následně můžeme zvolit parametr, který určuje počet prvků ze vstupu nutných na určení jednoho pivota. Tedy pokud tento parametr určíme jako d, pak počet prvků nutných k nalezení i pivotů je roven $d \cdot i$. Uvedený způsob byl využit v naší implementaci řadícího algoritmu.

Jedním z dalších přístupů je určování pivota tzv. dynamicky, jak doporučují autoři článku [10]. Hlavní myšlenkou je udržování statistik, jako je počet prvků v levém, resp. pravém segmentu a jejich součet. Na konci rozdělování se spočte průměrná hodnota, které prvky pro daný segment nabývají, a výsledná hodnota je využita jako pivot pro následující rekurzivní rozdělování. V práci autoři demonstrují dosažení vyšší efektivity v porovnání s přístupy Medián-tří a Medián-pěti. Problém nastává v obecnosti, jelikož autoři předpokládají, že jednotlivé elementy budou

podporovat základní aritmetické operace a tím se ubírá na obecnosti řadícího algoritmu. Jediné podporované operace pro zachování obecnosti je porovnání dvou prvků a jejich prohození. Vyžadování dodatečných operací by mohlo přinést omezení na vstupní data. Autoři dále neprovádějí rozsáhlejší testování v případě, kdy se ve vstupních datech vyskytuje veliké množství velmi malých elementů a malé množství velikých elementů. V takovém případě může být průměr velmi vzdálený mediánu, což je další z důvodů proč tento přístup výběru pivota nebudeme dále uvažovat.

2.2 Vícecestné rozdělování

V předchozích sekcích jsme si uvedli algoritmy rozdělování v případě jednoho pivota a tím položili základ pro pochopení a probrání variant s vícero pivoty. První zmínka a implementace vícecestného rozdělování pochází z roku 1975 z doktorandské práce Roberta Sedgewicka. [11] Jeho metoda rozdělování okolo dvou pivotů byla uvedena jako efektivní řešení vstupů s vysokým počtem opakujících se prvků, tedy varianta *tlustého pivota*, kterou jsme si definovali v 2.1. Po analýze z hlediska počtu porovnání, prohození elementů a počtu přečtených prvků jím byl tento přístup označen jako neefektivní. Celkově se na vícecestné rozdělování (2 a více pivotů) nahlíželo v dvacátém století velmi skepticky, jelikož teoretické analýzy algoritmu z hlediska počtu prohození elementů dosahovaly horších výsledků. [6]

V roce 2009 byla uvedena nová implementace řadícího algoritmu v jazyku Java, která dosahovala nižší doby běhu nežli předešlá implementace. Nový algoritmus *YBB* (Yaroslavskiy-Bentley-Bloch), nesoucí název po svých autorech, je implementací Quicksortu s rozdělováním pomocí dvou pivotů. Zrychlení se dosáhlo i navzdory faktu, že algoritmus dosahuje vyššího počtu prohození prvků a to skoro dvojnásobně oproti variantě s jedním pivotem 2.1. Naopak se dosáhlo nižšího počtu porovnání $1.9n \log n$ oproti implementaci s jedním pivotem $2n \log n$. [12] Analýza na čistě teoretické úrovni nám totiž nedá spolehlivý pohled na efektivitu algoritmu, jelikož nebere v potaz dnešní architekturu počítačových systémů, která má na rychlost algoritmu nezanedbatelný vliv.

Dosažené zrychlení bylo ještě větší, než by se dalo z počtu porovnání očekávat. Důvodem je již zmíněné opomenutí architektury moderních počítačových systémů a jejich paměťové hierarchie. Rozdíl mezi rychlostí paměti a CPU se v posledních letech čím dál tím více zvětšuje, a proto omezením přístupu do hlavní paměti na úkor náročnějšího výpočtu ze strany CPU a využitím cache pamětí může algoritmus dosáhnout značného zrychlení.

Tabulka 2.1 uvádí porovnání parametrů variant Quicksortu pro jednoho, dva (*YBB*) a tři pivoty (*Waterloo*). Parametr, ve kterém se algoritmy liší, je multiplikativní konstanta. Asymptoticky jsou však parametry pro všechny varianty identické, což plyne z povahy řadících algoritmů. Parametr *Cache Miss* udává počet čtení/zápisu elementů, které se nenacházely v cache paměti procesoru a bylo nutné je načíst z hlavní paměti. To má za následek výrazné zpomalení algoritmu, jelikož je procesor zdržován čekáním na data potřebná k výpočtu.

Můžeme si povšimnout, že z hlediska počtu prohození prvků je jako nejlepší algoritmus brán dvoucestný Quicksort. Obě vícecestné implementace naopak dosahují téměř dvojnásobného počtu prohození elementů vůči dvoucestné variantě, což však kompenzují počtem *Cache Miss* a nižším počtem porovnání. Varianta *Waterloo* by z tohoto pohledu měla být nejvíce efektivní. V neposlední řadě tyto algoritmy dosahují průměrně nižší hloubky rekurze, což je následkem skutečnosti, že v rámci jednoho rekurzivního volání odvedou „větší“ množství práce (data jsou rozdělena na tři nebo čtyři segmenty, oproti původním dvěma).

YBB rozdělování

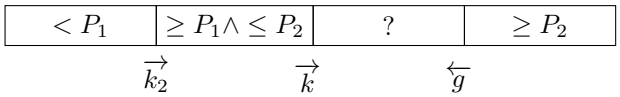
Jedná se o první implementaci vícecestného Quicksortu, která si našla místo v standardní knihovně jazyka Java. Pro rozdělování je využito $p = 2$ pivotů a rozděluje vstupní posloupnosti na tři segmenty. Pivoty před rozdělováním seřadíme a po skončení rozdělování bude první segment

Varianta	Cache Miss	Porovnání	Prohození
1-pivot	$2\frac{n+1}{B}\log\frac{n+1}{M+2}$	$2n\log n$	$0.333n\log n$
2-pivoti (*YBB*)	$1.6\frac{n+1}{B}\log\frac{n+1}{M+2}$	$1.9n\log n$	$0.6n\log n$
3-pivoti (*Waterloo*)	$1.38\frac{n+1}{B}\log\frac{n+1}{M+2}$	$1.85n\log n$	$0.62n\log n$

Tabulka 2.1 Porovnání parametrů různých variant Quicksort algoritmu s ohledem na počet pivotů. [12] M značí velikost cache paměti a B velikost jedné cache line.

obsahovat prvky menší než P_1, druhý segment prvky větší nebo rovny P_1 a prvky menší nebo rovny P_2. Poslední segment obsahuje prvky větší nebo rovny P_2. Hodnoty pivotů tedy volíme takovým způsobem, aby nám pole rozdělily na velikostně podobné třetiny. [6]

Průběh algoritmu je následující. Indexy k_2 a k začínají na stejném indexu v poli vlevo a postupně se přibližují k indexu g jdoucímu zprava. Pod indexem k se nachází nový prvek, který má být zařazen do správného segmentu. Ostatní indexy slouží jako oddělovače mezi postranními segmenty a vnitřním segmentem. Po zpracování všech elementů v oblasti označené „?“ se indexy k a g potkají a vytvoří konečnou hranici. Již ze schématu můžeme vidět silnou podobnost s dvoucestnými implementacemi rozdělování. Posun indexů k_2 a k je identický jako v případě *Lomuto* rozdělování a pouze se přidal dodatečný index g jdoucí zprava.



Obrázek 2.5 Invariant YBB rozdělování.

Pseudokód 2.4 zachycuje rozdělování pomocí YBB algoritmu. Zajímavou součástí algoritmu je funkce *CyclicShiftLeft*. Funkce vezme zvolené tři indexy do vstupních dat A a cyklicky prohodí prvky o jednu pozici vlevo. V algoritmu totiž nastává situace, která si žádá prohození vícero prvků najednou. K tomu dochází v případě, kdy jsou pod indexy k, k_2 a g postupně uloženy elementy, splňující $\geq P_2$, $\geq P_1$ a $< P_1$. Identického výsledku by se dalo dosáhnout sérií dvou po sobě jdoucích prohození, ale přístup využívající cyklický posun je efektivnější.

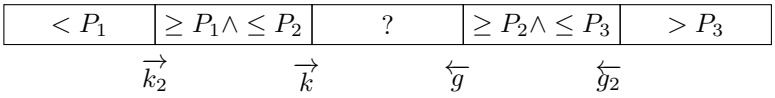
Výpis kódu 2.4 Pseudokód YBB rozdělování.

```
PartitionYBB(A, lp, rp)
    pi1, pi2 = GetPivot(A, lp, rp)
    swap(A[pi1], A[lp])
    swap(A[pi2], A[rp])
    p1 = A[lp]
    p2 = A[rp]
    k2 = lp + 1, k = k2, g = rp - 1
    while (k <= g)
        if (A[k] < p1)
            swap(A[k2], A[k])
            k += 1
        else
            if (A[k] >= p2)
                while (k < g && p2 < A[g])
                    g -= 1
                if (A[g] >= p1)
                    swap(A[k], A[g])
                else
                    CyclicShiftLeft(k, k2, g)
                    k2 += 1
                g -= 1
        k += 1
    swap(A[rp], A[g + 1])
    swap(A[lp], A[k2 - 1])
    return k2 - 1, g + 1
```

Waterloo rozdělování

V tomto typu rozdělování uvažujeme počet pivotů $p = 3$, kdy se vstupní posloupnost rozdělí na čtyři segmenty a jedná se tedy o čtyřcestné rozdělování. Po ukončení rozdělování bude první segment obsahovat prvky $< P_1$, druhý segment $\geq P_1 \wedge \leq P_2$, třetí $\geq P_2 \wedge \leq P_3$ a poslední $> P_3$. Algoritmus byl publikován v roce 2013 na univerzitě Waterloo, odkud plyne jeho název. [6] [12] Dle tabulky 2.1 dosahuje nejvyšší efektivity z hlediska práce s cache pamětí, a proto by měl být teoreticky rychlejší než algoritmus YBB.

Hlavním rozdílem, vyjma počtu pivotů, je druhý „vedoucí“ index g, který prochází vstupní posloupnost zprava doleva. Rozdělování skončí ve chvíli, kdy se indexy k a g potkají a vytvoří hranici mezi vnitřními segmenty. Obrázek 2.6 schématicky zachycuje průběh algoritmu.



Obrázek 2.6 Invariant Waterloo rozdělování.

Pseudokód 2.5 zobrazuje čtyřcestné Waterloo rozdělování. Můžeme si povšimnout větší symetričnosti vůči předchozímu algoritmu v podobě dvou vnořených while cyklů, kde indexy k a g

jdou proti sobě. Také nám přibylo množství cyklických posuvů, jelikož se nově vyskytují situace, kdy je třeba posunout elementy i vpravo. Navíc je nově potřeba provést posun čtyř elementů, protože pracujeme celkově se čtyřmi indexy. Všechny posuny by se daly opět implementovat jako série po sobě jdoucích prohození, ale z důvodu vyšší rychlosti posuvů byly nahrazeny.

Výpis kódu 2.5 Pseudokód Waterloo rozdělování.

```
PartitionWaterloo(A, lp, rp)
    pi1, pi2, pi3 = GetPivot(A, lp, rp)
    swap(A[pi1], A[lp])
    swap(A[pi2], A[lp + 1])
    swap(A[pi3], A[rp])
    k2 = k = lp + 2, g = g2 = rp - 1
    p1 = A[lp], p2 = A[lp + 1], p3 = A[rp]
    while (k <= g)
        while (k <= g && A[k] < p2)
            if (A[k] < p1)
                swap(A[k2], A[k])
                k2 += 1
            k += 1
        while (k <= g && p2 < A[g])
            if (p3 < A[g])
                swap(A[g], A[g2])
                g2 -= 1
            g -= 1
        if (k <= g)
            if (p3 < A[k])
                if (A[g] < p1)
                    CyclicShiftRight(k2, k, g2, g)
                    k2 += 1
                else
                    CyclicShiftLeft(k, g, g2)
                k += 1, g -= 1, g2 -= 1
            else
                if (A[g] < p1)
                    CyclicShiftRight(k2, k g)
                    ++k2;
                else
                    swap(A[k], A[g])
                k += 1, g -= 1
    k2 -= 1, k -= 1, g += 1, g2 += 1
    swap(A[lp + 1], A[k2]), swap(A[k2], A[k])
    k2 -= 1
    swap(A[lp], A[k2]), swap(A[rp], A[g2])
    return k2, k, g2
```

2.3 Vyhodnocení a výběr rozdělování

Součástí paralelního vícecestného Quicksort algoritmu bude jeho sekvenční verze. Paralelizace dává smysl pro rozsáhlé posloupnosti, a proto se na kratší využívá sekvenční varianta, jelikož nepřináší zpomalení způsobené managementem vláken. V sekcích 2.1 a 2.2 jsme si probrali možné

typy rozdělování po jejich teoretické stránce, jejich možné implementace a vlastnosti. V této sekci naopak prakticky změříme jednotlivé ukazatele, porovnáme je s těmi teoretickými (viz. tabulka 2.1) a po vyhodnocení zvolíme tu nejlepší sekvenční implementaci.

Byly implementovány tři Quicksort varianty výše s rozdělováním *Hoare*, *YBB* a *Waterloo*. Tyto algoritmy budeme porovnávat z hlediska počtu operací prohození (swap), porovnání (compare), doby řazení a počtu cache miss. Díky kombinaci těchto parametrů jsme schopni algoritmy vůči sobě spolehlivě porovnat a vyhodnotit. Vstupní data jsou datového typu *int* a jsou generována náhodně z uniformního rozdělení. Velikost vstupních dat byla zvolena jako 10^7. Díky náhodným datům a jejich délce bychom měli potlačit nevhodný výběr pivota a zvýhodnění některého z algoritmu. Ač byla data náhodně generována, jednotlivé algoritmy pracují nad stejnými daty, čímž zamezíme nekonzistencím v měření.

Pro přesné měření počtu operací prohození a porovnání všechny tři varianty využívají stejný algoritmus pro výběr pivota – *Medián-tří*. Pro varianty *YBB* a *Waterloo* je algoritmus výběru mírně poupraven. V případě YBB je nutný výběr dvou pivotů a proto si z vybraných tří prvků nezvolí jejich medián, ale první a poslední prvek. Waterloo algoritmus volí jako pivoty všechny tři seřazené prvky. Díky tomu do těchto dvou algoritmů nezaneseme vyšší počet operaci než je potřeba. Zároveň jsme pro měření zamezili optimalizacím pro velmi krátké posloupnosti a nebo při překročení hloubky rekurze při nevhodné volbě pivota.

Tabulka 2.2 prezentuje naměřené počty prohození a porovnání jednotlivých algoritmů. Pro přehlednost a zjednodušení vyhodnocení jsou hodnoty převedeny na poměr mezi algoritmem *Hoare*, který je brán jako výchozí, a algoritmem YBB resp. Waterloo. Pokud tedy vyjadřujeme empiricky naměřený poměr v případě *Hoare* algoritmu, vyjde nám počet porovnání a prohození 1. Pro viditelnější porovnání vůči teoretickým hodnotám byly empirické hodnoty porovnání vynásobeny koeficientem 2 a empirické hodnoty prohození koeficientem 0.333. Z tabulky jasně vidíme, že empirické hodnoty skoro odpovídají teoretickým. V případě porovnání oba algoritmy dokonce dosáhly lepších výsledků, než se předpokládalo. Naměřené počty prohození pro algoritmus Waterloo vyšly o dvě setiny hůře nežli teoretické. Empiricky jsme tedy potvrdili, že vícecestné rozdělování přináší nižší počet porovnání vůči dvoucestnému, naopak skoro dvojnásobný počet prohození.

	Porovnání		**Prohození**	
Varianta	**Teoretické**	**Empirické**	**Teoretické**	**Empirické**
Hoare	2	2	0.333	0.333
YBB	1.9	1.827	0.6	0.554
Waterloo	1.85	1.701	0.62	0.643

Tabulka 2.2 Porovnání teoretických hodnot (pouze koeficienty) z tabulky 2.1 s empiricky naměřeným počtem porovnání a prohození.

Dále bylo třeba zjistit počet cache miss jednotlivých implementací. Pomocí nástroje *PAPI* jsme schopni tuto vlastnost algoritmu měřit na úrovni L1, L2 a L3 cache. [13] Nástroj nabízí množství parametrů k měření, ale ne všechny musí být na daném zařízení k dispozici. Tato skutečnost při měření způsobila problémy v případě clusteru STAR (popsán v sekci 4.1). Na výchozím uzlu jsou totiž k dispozici jiné parametry než na výpočetním uzlu, kde probíhají jednotlivá měření a spouštění aplikace jako takové. To způsobovalo chyby v měření a bylo třeba zjistit dostupné parametry přímo na výpočetním uzlu. Z dostupných parametrů byly vybrány *PAPI_L1_DCM*, *PAPI_L2_DCM* a *PAPI_L3_TCM*. První dva parametry uvádějí počet cache miss při přístupu do datové cache paměti úrovně L1 a L2. V rámci měření nebylo možné uvést poměr vůči celkovému počtu přístupů do těchto pamětí, jelikož parametry nejsou dostupné a nelze je měřit. Z toho důvodu jsou uvažovány pouze cache miss a jednotlivé implementace porovnávány jen na základně tohoto parametru. PAPI_L3_TCM pro cache úrovně L3 udává počet cache miss pro instrukční a datovou cache dohromady. Měření pouze datové části opět nebylo k dispozici,

jelikož je tato úroveň sdílena jak pro data, tak pro instrukce programu.

Tabulka 2.3 obsahuje výsledky měření parametru cache miss jednotlivých algoritmů v různých úrovních cache. Algoritmus *Hoare* je opět brán jako referenční a hodnoty jsou uváděny jako poměr mezi referenčním algoritmem a algoritmy YBB a Waterloo. Z toho důvodu je teoretický koeficient roven empiricky naměřeným hodnotám. Poměr v případě *Hoare* algoritmu opět vychází jako 1, a proto byly pro lepší porovnání s referenční tabulkou 2.1 všechny empiricky naměřené hodnoty vynásobeny koeficientem 2. Z výsledku je jasně patrné, že oba přístupy k vícecestnému rozdělování dosahují lepší časové a prostorové lokality vůči dvoucestné variantě *Hoare*, na všech úrovních cache. Situace, kdy počet cache miss mezi jednotlivými úrovněmi je v nerovnosti $L1 \leq L2 \leq L3$, nastává u algoritmů, které obecně disponují velmi dobrou časovou a prostorovou lokalitou. Algoritmy na malé podmnožině vstupu odvedou majoritní množství práce a cache miss zpravidla nastává v případě, kdy začínají zpracovávat další část vstupu určeného k rozdělování a ten je potřeba načíst z hlavní pamětí. Pokud porovnáme průměr cache miss ze všech tří úrovní pro jednotlivé algoritmy s teoretickým koeficientem, tak vidíme, že mají odchylku pouze v rámci setin. Empiricky naměřené hodnoty vyšly ještě lépe, než jsme předpokládali a algoritmus Waterloo vychází jako nejlepší z hlediska přístupu do cache pamětí.

Varianta	Teoretické koeficienty	L1	L2	L3	Průměr
Hoare	2	2	2	2	2
YBB	1.6	1.464	1.492	1.723	1.560
Waterloo	1.38	1.270	1.297	1.425	1.331

Tabulka 2.3 Porovnání teoretických hodnot (pouze koeficienty) z tabulky 2.1 s empiricky naměřenými počty cache miss v jednotlivých úrovních cache.

Čas doby běhu algoritmu měříme pomocí knihovny *GoogleBenchmark*, která byla využita i na vyhodnocení paralelního Quicksort algoritmu a na dílčí měření v průběhu implementace a ladění parametrů algoritmů. Měření doby běhu probíhala na clusteru STAR, jehož specifikace je uvedena v kapitole 4. Doba je uváděna s přesností na milisekundy.

Tabulka 2.4 uvádí dobu běhu jednotlivých algoritmů a jejich procentuální zrychlení vůči výchozí variantě *Hoare*. Doba běhu koresponduje s předešlými tabulkami, kdy vícecestné algoritmy vycházely z hlediska počtu operací prohození, porovnání a počtu cache miss lépe než dvoucestná varianta. Zrychlení algoritmu Waterloo je vůči *Hoare* téměř 8% a vychází jako nejlepší z uvažovaných algoritmů.

Varianta	Čas	Zrychlení
Hoare	13 574 ms	0 %
YBB	12 925 ms	5.02 %
Waterloo	12 579 ms	7.91 %

Tabulka 2.4 Porovnání doby běhu jednotlivých algoritmů na náhodné posloupnosti generované z uniformního rozdělení.

Provedli jsme analýzu různých algoritmů pro vícecestné rozdělování a porovnali je mezi sebou na základě rozsáhlého množství parametrů a porovnali je s předpokládanými teoretickými hodnotami. Na základě empirické analýzy vychází z uvažovaných algoritmů Waterloo rozdělování jako nejlepší způsob v rámci řazení dat algoritmem Quicksort. Z toho důvodu bude jeho implementace využita v paralelní verzi Quicksort algoritmu pro sekvenční seřazení méně rozsáhlých úseků.

2.4 Kombinace s ostatními řadícími algoritmy

Paralelizace Quicksort algoritmu přináší zrychlení oproti sekvenční variantě pouze od určitého počtu vstupních elementů. Pokud tuto hranici nepřesáhneme, tedy budeme chtít seřadit vstup kratší než námi zvolená mez, seřadíme daný vstup pomocí sekvenčního řadícího algoritmu. Tato mez je stanovena empiricky na základě měření doby běhu na různý datových typech a seřazenosti vstupní posloupnosti.

Pokud bychom paralelní verzi algoritmu využívali i na krátké posloupnosti, dosáhli bychom paradoxně zpomalení než zrychlení vůči sekvenční verzi. To plyne ze skutečnosti, že režie vláken začne být velice časově nákladná a paralelizace se již nevyplatí. K přepnutí na sekvenční verzi může dojít při již zmíněné příliš malé vstupní posloupnosti a nebo při hluboké rekurzi. K přílišné hloubce rekurze může dojít, pokud dochází k nevýhodnému výběru pivotů (nebo v případě velkého počtu identických elementů) a posloupnost po rozdělování není rozdělena na poměrově stejné části, které vstupují do následujícího rekurzivního volání. V nejhorším případě by se mohlo stát, že by v případě tří-cestného rozdělování vznikl pouze jeden segment, který vstupuje do následujících rekurzivních volání Quicksortu. Pokud bychom hloubku rekurze nehlídali, mohl by takový průběh algoritmu vést až k úplné degradaci časové složitosti, což je $O(n^2)$.

Proto bude paralelní implementace kombinována se sekvenčním algoritmem využívajícím Waterloo rozdělování. Kromě zavolání sekvenčního řadícího algoritmu se samotný sekvenční Quicksort zpravidla kombinuje s *Heapsort* a *Insertion sort*. To vedlo k vytvoření tzv. *Heap-Insertion sort* a jeho využití v rámci sekvenčního čtyřcestného quicksort algoritmu. [5] Empiricky nám mez pro spuštění sekvenčního řadícího algoritmu v paralelní verzi vyšla na hodnotu mezi 100 000 a 500 000 elementy a finální hodnota byla stanovena na $1 << 18 = 262\ 144$ prvků (operátor $<<$ značí bitový posun doleva).

Heapsort

Přepnutí na Heapsort algoritmus se využívá v případech, kdy hrozí degradace na kvadratickou složitost Quicksort algoritmu v případě neúměrné hloubky rekurze. [9] K tomu dochází při vytváření nepoměrových velikostí částí po rozdělování, jak bylo popsáno v předešlém textu. U Heapsort algoritmu máme totiž vždy zaručenou složitost $\Theta(n \log n)$. Algoritmus je in-place, datově necitlivý a nerekurzivní, což v tomto případě kompenzuje nedostatek Quicksortu.

Parametr hloubky rekurze u sekvenčního vícecestného Quicksortu empiricky nejlépe vycházel při nastavení $depth = 1.5 \log_4 n$. V naší implementaci se vstupní posloupnost dělí na 4 části, proto je využit logaritmus o základu 4 a ne 2, jako je tomu v implementaci STL knihovny. Parametr je v algoritmu *std::sort* od GNU nastaven na $depth = 2 \log_2 n$.

Insertion sort

Tento algoritmus se ve standardní implementaci využívá na řazení malých posloupností. Je tomu tak proto, jelikož nabízí pozitivní vlastnosti u krátkých posloupností, jako je paměťová lokalita (časté čtení a zápis elementů, které jsou v paměti blízko u sebe) a předvídatelnost pro branch predictory. Stejně jako *Heapsort* (popsaný výše) je algoritmus in-place a nerekurzivní. Fakt, že není rekurzivní u malých posloupností, šetří časté volání funkce a ukládání/čtení proměnných ze zásobníku. Rozdílem je jeho datová citlivost na rozdíl od Heapsortu.

Heap-Insertion sort

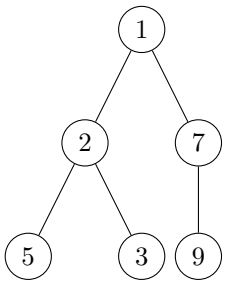
Při testování efektivnosti algoritmu jsem zjistil, že je implementace Heapsort algoritmu neefektivní a není zřejmě určena k řazení (především funkce *std::make_heap*). To vedlo na implementaci algoritmu *Heap-Insertion sort* dle [5]. Hlavní myšlenkou je zkombinovat vlastnosti a sílu

obou výše uvedených algoritmů a tuto verzi volat v případě, kdy sekvenční vícecestný Quicksort přesáhne hloubku rekurze a nebo když vstupní posloupnost dosahuje délky menší než námi stanovená mez. Hlavní myšlenkou tohoto algoritmu je upravit data takovým způsobem, aby byla vhodná k řazení pomocí Insertion sortu. Toho se dá docílit pomocí vytvoření binární haldy a následného zavolání Insertion sort na takto předzpracovaná data. Tím bude algoritmus nucen procházet méně prvků a počet prohazování elementů se sníží.

5	3	7	2	1	9

Obrázek 2.7 Vstupní posloupnost určená k seřazení pomocí Heap-Insertion sort algoritmu.

Obrázek 2.7 zobrazuje pole o šesti prvcích, které má být seřazeno. Pozice prvků jsou náhodné a na jejich seřazení pomocí Insertion sort je třeba 8 prohození. Po vytvoření binární minimální haldy nám vznikne strom, zobrazený na 2.8. Můžeme si povšimnout vlastnosti, že cesta z listu do kořene tvoří klesající seřazenou posloupnost.



Obrázek 2.8 Minimální binární halda vytvořena pomocí funkce *make_heap* z pole.

Binární minimální halda uložena v poli je zachycena na obrázku 2.7. První element se dá považovat za již seřazený (plyne z povahy minimální haldy) a ostatní prvky jsou na výhodnějších pozicích pro Insertion sort. Algoritmus v tomto případě provede pouze 3 prohození, než bude pole celkově zpracováno.

Mez pro spuštění algoritmu Heap-Insertion sort v rámci sekvenčního Quicksortu byla empiricky stanovena na 64 prvků a hloubku rekurze popsanou v 2.4.

1	2	7	5	3	9

Obrázek 2.9 Posloupnost po zavolání funkce *make_heap* určená k seřazení Insertion sort algoritmem.

Kapitola 3
Paralelní Quicksort

Během rešeršní části práce nebyla nalezena implementace vícecestného paralelního Quicksortu. Všechny dosavadní paralelní verze využívají v paralelním rozdělování pouze jednoho pivota a jsou tedy dvoucestné. Pohled na sekvenční verze a analýzu vícecestných variant v kapitole 1 přináší motivaci pro implementaci vícecestné varianty i ve vícevláknovém prostředí se sdílenou pamětí, jelikož vícecestné varianty vždy přinesly zrychlení vůči původní dvoucestné. Kvůli neexistenci vícecestné paralelní verze nebylo možné navázat na předešlé práce a tím je tato práce unikátní a originální.

V této kapitole si probereme původní dvoucestnou verzi a přístup, který k paralelnímu rozdělování zastává. To nám umožní pochopit základní principy a metody, jak lze rozdělování efektivně a škálovatelně paralelizovat. Následně probereme možnosti aplikace v případě většího množství pivotů a zda by se dal tento přístup využít, či jaké modifikace budou nutné pro vícecestnou implementaci. Cílem je vytvořit paralelní vícecestnou in-place variantu algoritmu. V textu probereme různé uvažované přístupy, z nichž byl vybrán jeden, který umožnil předejít limitacím vícecestné verze a umožnil její implementaci. Ostatní uvažované přístupy by již z teoretické analýzy nepřinesly zrychlení. Nový algoritmus bude vysvětlen, jednotlivé části popsány a také uvedeme dodatečné změny v implementaci po průběžných měřeních rychlosti algoritmu.

Důležitou součástí paralelních algoritmů je vyvažování zátěže rovnoměrně mezi jádra. Chceme tedy minimalizovat případy, kdy by skupina jader intenzivně pracovala na výpočtu a druhá skupina jader by neměla na čem pracovat. Pokud by tento případ nastal, vedlo by to k neefektivitě algoritmu a špatné škálovatelnosti v případě narůstajícího počtu jader. Jádrem myslíme fyzické jádro na CPU, pro naše účely nebudeme uvažovat hyper-threading. Hyper-threading je funkcionalita procesoru, kdy jedno fyzické jádro využívají dvě virtuální. Funkcionalita je běžnou součástí moderních procesorů, ale kvůli sdílení cache pamětí mezi virtuálními jádry ji nebudeme v textu uvažovat.

Pro paralelizaci byla využita knihovna OpenMP [14], která se běžně využívá a poskytuje přehlednou a jednoduchou implementaci paralelizace algoritmů s minimálním množstvím dodatečného zdrojového kódu.

3.1 Existující dvoucestné varianty

Paralelní dvoucestná varianta Quicksort vždy pracuje s jedním pivotem a rozděluje vstupní posloupnost do dvou segmentů. První segment obsahuje prvky menší než pivot a druhý segment prvky větší nebo rovny pivotu. V této sekci probereme pouze in-place variantu algoritmu. Existují implementace out-of-place, ale ty zde nebudeme uvažovat. Jedná se totiž o hlavní přednost algoritmu. Pokud se nejedná o nutný požadavek na algoritmus, je možné využít vícecestný paralelní

Mergesort.

V dnešní době nejpoužívanější varianta nese název F&A od autorů Tsigas a Zang, která je vylepšením varianty algoritmu od autorů Francis a Pannan. [15] [16] První z algoritmů byl představen na přednášce předmětu NI-PDP na ČVUT FIT a uveden jako paralelní in-place varianta algoritmu Quicksort. [7] Právě vylepšenou verzi F&A si zde probereme, uvedeme její teoretické principy a kroky vedoucí k efektivní paralelizaci. Pseudokód jednotlivých částí nebudeme uvádět, jelikož tato verze algoritmu nebyla implementována a slouží pouze jako seznámení se s problematikou paralelního řazení algoritmu Quicksort.

Paralelizace algoritmu Quicksort se provádí na vícero úrovních. První úrovní je paralelizace řídící části algoritmu. V této části se provede dvoucestné rozdělování a poté se algoritmus rekurzivně zavolá na oba nově vzniklé segmenty. Zde se rovnou nabízí paralelizace rekurzivních volání pomocí OpenMP konstruktu *task*. Nově vzniklý podproblém se vloží do tzv. *task poolu*, odkud si jednotlivé úlohy mohou brát ostatní vlákna a pracovat nezávisle na disjunktních podproblémech. Tato vlákna rekurzivně jednotlivé problémy řeší, vkládají do task poolu nové úlohy a zároveň si v případě, kdy svůj přidělený podproblém vyřešila, vezmou další. V řídící části algoritmu se také v případě klesnutí délky podposloupnosti pod určitou mez přepíná algoritmus na sekvenční verzi Quicksort algoritmu. Mez je určena empiricky na základě měření. Při generování nových úloh je vhodné rozdělit počet jader na základě velikosti vzniklých segmentů. Ač se snažíme vybrat pivota optimálně jako skoromedián, nejlépe přímo medián (viz. 2.1), může se i tak stát, že budou segmenty velikostně nevyrovnané. Proto je vhodné počet jader využitých k seřazení daného segmentu optimalizovat.

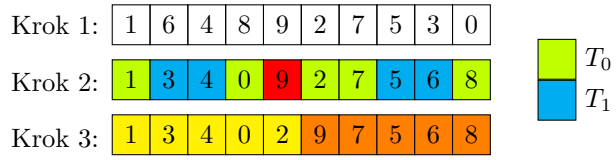
Výše popsaná paralelizace však není dostatečná a je třeba paralelizovat i další kroky algoritmu. Nejdéle trvající součástí algoritmu je rozdělování prvků do dvou segmentů a tato část je stále sekvenční. V první hladině stromu rekurzivních volání totiž nejsou v task poolu žádné úlohy pro ostatní jádra. Z toho důvodu by vstupní posloupnost pří prvním volání rozdělovalo pouze jedno jádro, které by díky optimalizaci koncové rekurze do task poolu vložilo pouze jednu úlohu a tedy by na další hladině měla práci pouze dvě jádra a ostatní by byla zcela nevyužitá. Již z tohoto popisu je jasně patrné, že by byl algoritmus velmi neefektivní a nevyužíval potenciálu CPU s více jádry. Rozdělování je jádrem algoritmu a jeho paralelizace je tedy nejvíce komplexní, pokud chceme dosáhnout co nejvyšší efektivity. [9] [17]

V sekci 2.1 jsme probrali dva zástupce dvoucestného rozdělování, jimiž byly algoritmy *Lomuto* a *Hoare*. Z hlediska paralelizace je mnohonásobně vhodnější algoritmus Hoare. Oba indexy algoritmu *Lomuto* postupují zleva doprava a jsou na sobě závislé. Závislostí rozumíme skutečnost, kdy index k_2 slouží jako oddělovač nově vznikajících segmentů a k určuje hranici konce druhého segmentu. Hranice začátku prvního segmentu je implicitně brána jako začátek celé posloupnosti. Pokud bychom algoritmus chtěli paralelizovat, musela by jednotlivá vlákna pracovat s identickými indexy (atomické operace) a nemohla by prvky zpracovávat v blocích, což vede k neefektivitě popsané dále v textu. U Hoare algoritmu indexy postupují proti sobě z levé, resp. pravé strany posloupnosti. Oba segmenty mají implicitní začátek, resp. konec určený jako začátek, resp. konec pole. Segmenty tedy rostou proti sobě, vlákna mohou na prvcích pracovat nezávisle a vždy mít své vlastní privátní indexy.

Po studiu Hoare algoritmu si všimneme skutečnosti, že vždy porovnáme prvky na indexech k a g, v případě nutnosti prvky prohodíme a inkrementujeme resp. dekrementujeme indexy dle potřeby. Tento proces nazýváme *neutralizace*. Pokud tedy docílíme, že každé vlákno pracuje s disjunktním párem indexů, můžeme prvky porovnávat a prohazovat paralelně. Každé vlákno své privátní indexy k a g získá pomocí operace OpenMP *atomic capture* ze sdílených indexů. Ta umožní atomicky získat původní hodnotu sdíleného indexu mezi vlákny, uložit ji do privátního indexu daného vlákna a atomicky ji inkrementovat pro ostatní vlákna. Pokud by operace inkrementace nebyla atomická, nebyl by algoritmus korektní z důvodu velmi pravděpodobného *race condition*. K němu dochází v případě, kdy více vláken zapisuje/čte do sdílené proměnné a získané páry indexů by nemusely být disjunktní. Algoritmus opět končí, pokud se indexy střetnou resp. $k \geq g$. V rámci paralelizace je ještě nutné provést *úklid*. Po skončení může mít každé vlákno je-

den tzv. *špinavý prvek*, který není ve správném segmentu. Index tohoto prvku má každé vlákno a vzhledem k jejich velmi malému počtu (nejvýše rovno počtu vláken) prvky umístíme na své místo sekvenčně. Tímto způsobem jsme paralelizovali i sekvenční část paralelního Quicksort algoritmu a získali dva segmenty.

Níže si můžeme prohlédnout jednoduché schéma algoritmu ve třech krocích. Prvky pole jsou rozlišeny barvou, abychom rozeznali, která z vláken je neutralizovala. Pivota volíme jako element s hodnotou 5. První krok zobrazuje vstupní posloupnosti deseti prvků určených k rozdělování. Druhý krok je po provedené neutralizaci prvků jednotlivými vlákny. Jedná se o jednu z mnoha konfigurací, jelikož vlákna mohou prvky získávat v různém pořadí. Toto pořadí je jedno z nich. V tomto případě vlákno T_0 skončilo neutralizaci prvků 2 a 7, které zůstaly na svých pozicích, tudíž hodnota sdílených indexů k a g je 6 resp. 5 a platí $k \geq g$. Z pohledu T_0 jsou všechny prvky neutralizovány a čisté. Naopak vláknu T_1 zbyl prvek 9, který byl brán zleva a je tedy nečistý. Krok 3 ukazuje výsledek rozdělování po úklidové fázi. Pivota je možné na začátku rozdělování umístit jako první resp. poslední element pole a při úklidové fázi jej přemístit na jeho finální pozici. Díky tomu nemusí element vstupovat do následujících rekurzivních volání a zrychlí se doba běhu.



Obrázek 3.1 Paralelní rozdělování dvoucestného Quicksort algoritmu po jednotlivých elementech.

Algoritmus popsaný v předešlém textu pracuje korektně, ale je neefektivní. Důvodem je *falešné sdílení*, ke kterému dochází při častém zápisu více vláken do stejného řádku cache paměti (*cache line*) a vysoký počet atomických operací. Nemusí se tedy ani jednat o identické místo v paměti, stačí, když dochází k zápisu do stejného řádku. Koherenční protokoly, jako například MESI, se starají o koherenci cache pamětí mezi vlákny a pracují právě v rámci celých řádků, nikoli bajtů. Pokud tedy vlákna T_0 a T_1 pracují nad daty, která jsou v rámci jednoho řádku, pak zápis do cache line ve vláknu T_0 způsobí invalidaci a opětovné načtení řádku v T_1, i když byla modifikována proměnná, se kterou T_1 vůbec nepracuje. V případě, kdy mají vlákna přidělena pouze dva prvky, bude k této situaci v průběhu algoritmu docházet neustále. I k atomickému čtení a zápisu do sdílených indexů dochází velmi často, což způsobuje dodatečné zpomalení. Řešením je vláknům namísto jednotlivých prvků přidělovat bloky o velikosti B (optimum určeno empiricky, zpravidla násobek velikosti cache line). O bloky si vlákna žádají z levého, resp. pravého konce pole. Prvky v přidělených dvou blocích neutralizují a namísto čistého bloku (blok obsahující pouze prvky, které do něj patří) si vlákno zažádá o nový a pokračuje v neutralizaci. Bloky jsou přidělovány identicky jako jednotlivé indexy, tedy pokud je pravý blok označen jako čistý, vlákno atomicky získá hodnotu sdíleného indexu g a jeho hodnotu dekrementuje o velikost B. Díky blokovému přístupu silně redukujeme počet falešných sdílení a počet atomických operací nad sdílenými indexy k a g.

Stejně jako u přidělování jednotlivých indexů vláknům, i zde mohou vzniknout tentokrát tzv. *špinavé bloky*. Blok označíme jako špinavý v případě, kdy pro sdílené indexy platí $k \geq g$ a blok obsahuje prvky z dvou segmentů. Počet špinavých bloků může být opět roven nejvýše počtu vláken. Tyto špinavé bloky je opět nutné neutralizovat a prvky umístit do správných segmentů. Vzhledem k tomu, že může být prvků veliké množství, je vhodně provést úklid paralelně. Implementaci neutralizace špinavých bloků nebudeme dále rozebírat, jelikož pro nás není stěžejní.

Tímto způsobem jsme docílili efektivní paralelizace části dvoucestného rozdělování algoritmu Quicksort. Díky tomu víme, kde algoritmus stráví nejvíce času a na co musíme klást při návrhu a implementaci vícecestného paralelního Quicksort algoritmu důraz. Práci musíme vláknům přidělovat v blocích, nikoli po jednotlivých elementech. Velikost bloku by měla být optimálně

násobkem velikosti cache line. Je nutné docílit vyvažování zátěže mezi jednotlivá vlákna a práci se pokusit rozdělit co možná nejvíce rovnoměrně. Po označení bloku jako čistého víme, že jeho prvky na daném indexu zůstávají a jsou tedy ve správném segmentu. Z toho důvodu není nutný jejich další přesun v rámci aktuálního rekurzivního volání. Počet prohození elementů v rámci rozdělování je tedy minimální a měli bychom se v následující implementaci snažit o to samé. Na vyšších hladinách rekurze je volba pivota mnohonásobně důležitější než v hlubších hladinách. Ve vyšších hladinách je posloupnost rozsáhlejší, a proto má nevyváženost velikosti segmentů významnější dopad na rychlost algoritmu.

3.2 Analýza možností paralelizace

Vůči dvoucestnému paralelnímu rozdělování a sekvenčnímu vícecestnému rozdělování přináší paralelizace s počtem pivotů $p > 1$ řadu obtíží, které bylo třeba vyřešit a navrhnout korektní algoritmus a postupně jej optimalizovat. Bylo uvažováno několik možných variant a návrhů algoritmů, z nichž byl vybrán jeden a ten byl následně implementován. Ostatní varianty byly po analýze neefektivní a nebo nebyl nalezen způsob, jak paralelizace vůbec docílit. Uvažované varianty a návrhy zde probereme, uvedeme důvody proč nebyly vybrány a nakonec důkladně popíšeme výslednou implementaci, včetně jednotlivých optimalizačních kroků.

Na první problém narazíme, pokud bychom chtěli vícecestné rozdělování paralelizovat obdobně, jako u klasického paralelního rozdělování v 3.1. Z povahy uložení vstupu a toho, že se elementy rozdělují do dvou segmentů, vyplývají implicitně dobré možnosti paralelizace a minimální přesun elementů. Ze skutečnosti, že bude pole rozděleno do dvou částí přesně víme, odkud si mají jednotlivá vlákna elementy (případně bloky) brát a kam je mají vkládat. Začátek resp. konec vstupního pole implicitně označuje začátek prvního segmentu s_0 a konec segmentu s_1. Informace o přesné hranici rozdělení dvou segmentů není na začátku algoritmu vůbec třeba. Finální rozdělení segmentů se určí v úklidové fázi a v průběhu neutralizace bloků jej není třeba uvažovat. Všechna vlákna mají výchozí indexy, na základě kterých si mohou vzít levý a pravý blok. Po označení bloku jako čistého víme, že prvky v daném bloku již nebude nutné v rámci tohoto rozdělování přesunout a jsou tedy ve správném segmentu.

Pokud však chceme provést vícecestné rozdělování pro $p > 1$, pak nám implicitní hranice tvořené začátkem a koncem vstupní posloupnosti nestačí. Pokud budeme pro jednoduchost uvažovat počet pivotů $p = 3$, pak máme za cíl posloupnost rozdělit do čtyř segmentů (viz. 2.2). Tento algoritmus je symetrický a jedná se o kombinaci Hoare a *Lomuto* rozdělování. Nastává zde obdobný problém, kvůli kterému byl v případě dvoucestného rozdělování zvolen algoritmus Hoare. Na základě pochopení a uvedení implementace 3.1 víme, že je nutné vláknům přidělovat bloky elementů vstupní posloupnosti, nikoli jednotlivé elementy. Víme, kde bude začátek resp. konec krajních segmentů s_0 resp. s_3. Nevíme však, kde začínají nebo končí vnitřní segmenty s_1 a s_2. Pokud bychom pro jednoduchost předpokládali perfektní nalezení tří pivotů, které rozdělí posloupnosti na čtyři segmenty ekvivalentních velikostí, pak bychom věděli, kde je konec segmentu s_1 a začátek segmentu s_2. Počet segmentů musí být tedy vždy $s = p + 1$, aby mohl vzniknout alespoň jeden neutralizovaný blok pro dané vlákno. Takové rozdělení je však krajně nepravděpodobné a v reálném prostředí s takto ideálním rozdělením nemůžeme počítat. Mohli bychom si tedy udržovat počet čistých bloků spadajících do každého segmentu a na základě toho vyvažovat výběr bloků. I to by však neumožnilo efektivní rozdělování do segmentů. V extrémní situaci v případě častého opakování elementů by se mohlo stát, že by byla například délka s_1 nulová a délka s_0 velmi malá s porovnáním délek s_2 a s_3. V takovém případě by segmenty s_2 a s_3 „rostly“ proti sobě. V určité fázi bychom zjistili, že se prvky segmentu s_3 nemají kam vkládat a bylo by nutné prvky z s_2 přesouvat znovu, abychom uvolnili prvkům v s_3 místo v poli.

Problémová situace popsaná výše by mohla nastávat velmi často a teoreticky by mohly nastat ještě horší situace, což by způsobilo silnou neefektivitu algoritmu. Po analýze dvoucestné varianty víme, že po neutralizaci je prvek v příslušném segmentu a není třeba jej nadále v rámci daného rozdělování přesouvat. Tato vlastnost by v tomto případě nebyla dodržena.

Dále víme, že *Lomuto* algoritmus není možné efektivně modifikovat takovým způsobem, abychom zajistili přidělování elementů jednotlivým vláknům v blocích. Indexy k_2 a k resp. g_2 a g jsou na sobě v algoritmu Waterloo obdobně závislé. Indexy k_2 a g_2 dynamicky vytváří hranici mezi vnitřním a krajním segmentem za běhu algoritmu rozdělování. V případě sekvenční implementace není dynamické určování hranice problém, ale v případě vícevláknového prostředí nejsme schopni vytvořit disjunktní indexy bloků a ty přidělit jednotlivým vláknům. Tento přístup by vždy vedl na dodatečná prohození elementů při neznalosti hranic segmentů.

Dalším nápadem pro vícecestnou paralelizaci bylo využití vícecestného Mergesort algoritmu, konkrétně jeho paralelní implementaci p-cestného slučování. Vstupní posloupnost by se rozdělila do c částí, kde c představuje počet vláken resp. jader procesoru. Na každou část by se paralelně zavolal sekvenční vícecestný Quicksort algoritmus. Tím bychom získali c seřazených částí, které by bylo nutné slít dohromady a získat jednu seřazenou posloupnost. Také bychom se vyhnuli paralelizaci části rozdělování a dosáhli vyváženosti zátěže mezi vlákna (každé vlákno by mělo přibližně stejně velikou část pole ke zpracování). Dále by nebylo třeba synchronizačních primitiv, kromě bariéry na konci paralelního seřazení c částí. Ač tento přístup vyřešil náš problém s velikostí segmentů, způsobil problémy jiné. Prvním z nich je, že efektivní implementace paralelního slučování je out-of-place. Velmi silnou a žádanou vlastností Quicksort algoritmu je právě skutečnost, že k běhu nepotřebuje další paměť a je tedy in-place (1.2). O tuto vlastnost nechceme přijít, jelikož je v takovém případě někdy výhodnější využít rovnou paralelní Mergesort algoritmus. Existují i in-place varianty, ale ty jsou zpravidla časově neefektivní. Posledním důvodem vyřazení tohoto způsobu pro řazení při návrhu algoritmu je skutečnost, že se pak v podstatě nejedná o vícecestný paralelní Quicksort algoritmus. Sice se jednotlivé bloky určené pro každé vlákno řadí pomocí vícecestného Quicksortu paralelně, ale hlavní část by se v podstatě řešila využitím zcela jiného algoritmu. Z těchto důvodů jsme byli nuceni vyloučit i tento přístup.

Po zkoumání dalších přístupů a možností paralelizace rozdělování jsme došli k závěru, že se bez zjištění velikosti segmentů a získaní jejich hranic paralelizace vícecestného Quicksort algoritmu provést nedá. Pokud tedy v rámci rozdělování vstupní posloupnost projdeme dvakrát, kdy si v rámci prvního průchodu napočítáme velikosti jednotlivých segmentů, můžeme z nich vláknům přiřazovat disjunktní bloky, které mohou nezávisle neutralizovat. Tím docílíme, že po neutralizaci bloku jsou jeho elementy ve správném segmentu a není nutné jejich dodatečné prohazování. Provádět průchod dvakrát v rámci každé rekurze by však vedlo k neefektivitě. Zároveň se ale zjištění velikosti jednotlivých segmentů nedá vyhnout. Z toho důvodu chceme provést počítání velikostí segmentů pouze jednou a zbytečně tímto výpočtem neprodlužovat dobu běhu následujících rekurzivních volání. Z důvodu vyvažování zátěže mezi vlákna je rozdělování na čtyři segmenty pomocí $p = 3$ pivotů nevýhodné a ostatní vlákna by hladověla, v případě procesorů s více než čtyřmi jádry. Proto zvolíme velikost p tak, abychom vždy získali dostatečný počet segmentů, který je optimálně roven počtu přidělených jader.

Díky návrhu výše dodržíme vlastnosti zjištěné u dvoucestné varianty. Práci můžeme jednotlivým vláknům přidělovat blokově, zátěž bude rozložena mezi vlákna (detaily v následujícím textu) a elementy v čistém bloku již není třeba dodatečně přesouvat (jen zanedbatelné množství).

Popsaný přístup vede na implementaci vícecestného paralelního Quicksort algoritmu, popsaného v této práci, s názvem *MPQsort* (*Multi-way Parallel Quicksort algoritm*). Jedná se o čistou implementaci s využitím základních konstruktů algoritmu Quicksort, bez kombinace s jinými řadicími algoritmy (kromě optimalizací pro krátké posloupnosti) a prvním algoritmem svého druhu.

Paralelizace využitím OpenMP

OpenMP je knihovna pro jazyky Fortran a C/C++, která umožňuje přehlednou paralelizaci algoritmů. Pro paralelizaci mohou být využity poskytované funkce knihovny a nebo direktivy *#pragma omp* pro preprocesor resp. kompilátor. To umožňuje paralelizaci programů na vyšší úrovni abstrakce a nemusíme se nutně zabývat konkrétní implementací vláken a synchronizačních

primitiv daného jazyka. [14]

Knihovna využívá tzv. *fork-join* model, kdy se původně čistě sekvenční kód rozdělí do série paralelních regionů. Na začátku regionu se vytvoří požadovaný počet vláken (fork), která budou region provádět paralelně, na konci se vlákna opět spojí (join) a kód je provádět opět sekvenčně. [7] Jednotlivé paralelní regiony jsou ohraničeny složenými závorkami a na konci mají vždy implicitní bariéru, kde rychlejší vlákna čekají na pomalejší a až poté je paralelní region ukončen. [14] Z důvodu optimalizace knihovna implementuje *thread pool*. Vytvoření vlákna má určitou režii, a proto se na konci paralelního regionu umístí do thread poolu. Pak je možné jej využít pro další výpočty. K destrukci vlákna dochází až při ukončení programu.

OpenMP nabízí iterační (datový) a funkční (task) paralelismus. Datový je využit v případě paralelizace cyklů, kdy se identické operace provádí nad disjunktní podmnožinou dat. Jednotlivé iterace jsou rozděleny mezi vlákna na základě specifikovaného plánování, které může mít výrazný vliv na dobu běhu. Pro paralelizaci cyklů je využita direktiva *#pragma omp parallel for*. Více informací o typech plánování je uvedeno v dokumentaci [14].

Funkční paralelismus, jinak také task, slouží pro paralelizaci částí kódu, které jsou na sobě nezávislé. Příkladem může být prohledávání binárního stromu, nebo právě paralelizace Quicksort algoritmu. Obecně se jedná o velmi vhodný konstrukt pro paralelizaci rekurzivních algoritmů. Pro využití funkčního paralelismu je využita direktiva *#pragma omp task*.

Při implementaci MPQsort byly využity oba zmíněné konstrukty, které dopomohly k paralelizaci algoritmu. Pro jejich kombinaci bylo nutné nastavit vnořený paralelismus, díky kterému je možno vytvářet „rekurzivní“ paralelní regiony.

U obou výše zmíněných konstruktů je možné specifikovat různé možnosti sdílení proměnných mezi vlákny, jejich inicializaci (hodnota pro dané vlákno) a také výstupní hodnotu. Nejvíce důležitými typy proměnných jsou *shared* (sdílené) a *private* (privátní). [7] [14] Sdílené proměnné jsou přístupné ze všech vláken, a proto je třeba přístup k nim ošetřit využitím synchronizačních primitiv, jako jsou atomické operace či mutex. Sdílenou proměnnou má naopak každé vlákno vlastní, a proto není třeba k nim přístup jakkoliv synchronizovat.

Nad diskuzí možností knihovny OpenMP bychom mohli strávit množství času. Nejdůležitější informace a konstrukty jsme zde však probrali a v případě zájmu může čtenář nahlédnou do výsledné implementace přiložené na CD, kde uvidí konkrétní parametry a využité konstrukty.

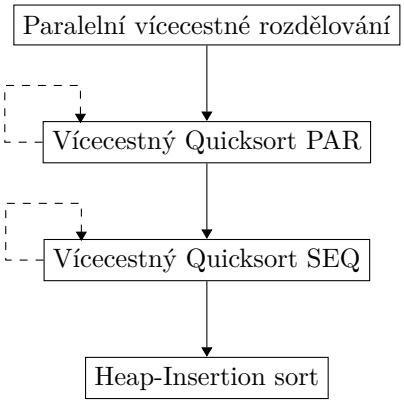
3.3 Návrh a implementace MPQsort

Hlavní motivací paralelizace části rozdělování v případě dvoucestné varianty je vyvažování zátěže na horních vrstvách stromu rekurzivních volání. Paralelizace rozdělování je nutností i v případě vícecestného přístupu, ale z důvodu nutnosti dvou průchodů posloupnosti bude probíhat v algoritmu *MPQsort* pouze jednou. V této fázi vytvoříme dostatečně velký počet úloh pro každé vlákno a předejdeme problému hladovění vláken. Tím odpadá nutnost paralelizace vícecestného rozdělování v rámci rekurzivních volání.

MPQsort se skládá ze čtyř fází, které zachycuje schéma 3.2. V první fázi provádíme paralelní vícecestné rozdělování, které je jádrem algoritmu. Probíhá zde příprava pro následující fáze, kdy vytvoříme s segmentů. Počet segmentů je v nejlepším případě roven počtu vláken. V ojedinělých případech se může stát, že jich bude menší počet. Implementace z důvodu efektivity zařazování elementů předpokládá, že počet segmentů bude 2^k, kde k je přirozeným číslem. Počet jader procesorů je běžně právě mocninou dvou, a tudíž v drtivě většině bude počet segmentů optimální. MPQsort umožňuje nastavením parametru modifikovat počet pivotů a tím zvýšit počet segmentů na $s > c$. Při měření v kapitole 5 jsme však došli k závěru, že takový počet není optimální.

Například cluster STAR má jader 20 a v takovém případě se počet segmentů určí jako nejbližší menší mocnina dvou. V tomto případě 16.

Po skončení vícecestného rozdělování v první fázi získáme s částí pole, které mohou být řazeny nezávisle na sobě. Funkce paralelního rozdělování nám vráti pole indexů s konci jednotlivých segmentů. Z těchto indexů si dopočítáme i jejich začátky. Díky těmto indexům víme velikosti



Obrázek 3.2 Schéma volání jednotlivých částí algoritmu MPQsort. Přerušovaná šipka zobrazuje rekurzivní volání a nepřerušovaná přechod mezi fázemi.

vytvořených segmentů a jejich počet. Z důvodu vyvažování zátěže indexy segmentů seřadíme na základě velikosti daného segmentu sestupně. Následně v paralelním for-cyklu s dynamickým plánováním voláme na jednotlivé segmenty algoritmus z druhé fáze. Pivoty se snažíme vybírat tak, abychom získali segmenty o co nejpodobnější velikosti, ale i přesto mezi nimi mohou být značné rozdíly. Proto je nejlepší začíst s tím největším, jelikož by měl zabrat nejdelší dobu. Pokud nějaké vlákno s přiřazenou částí skončí, vezme si další pár indexů a nebo úlohu vytvořenou jiným vláknem, dokud nejsou všechny části seřazeny a úlohy vyřešeny.

V druhé fázi je zavolán algoritmus *Waterloo* s paralelní řídící částí, využitím konstruktu *task* knihovny OpenMP. Vlákno provede sekvenční čtyřcestné rozdělování přiřazené části a získá čtyři segmenty. Pokud je velikost segmentu větší než empiricky určená mez, vloží tento segment do task poolu, odkud si mohou vytvořenou úlohu vzít ostatní vlákna. Jedná se o další způsob vyvažování zátěže, pokud by segmenty vytvořené algoritmem v první fázi nebyly velikostně vyvážené. Pro využití vnořeného paralelizmu v knihovně OpenMP je nutné jej zapnout pomocí *omp_set_nested(1)*, nebo nově nastavit povolenou hloubku vnořené rekurze pomocí *omp_set_max_active_levels(std::numeric_limits<int>::max())*. Pro jednoduchost jsme hloubku nastavili jako maximální možnou. Pokud velikost segmentu klesne pod sekvenční mez, přechází algoritmus do třetí fáze.

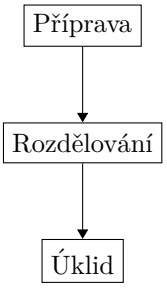
Ve třetí fázi je využita klasická implementace Quicksort algoritmu s Waterloo rozdělováním. Algoritmus je plně sekvenční a slouží pro řazení kratších polí, v jejichž případě paralelizace naopak způsobuje zpomalení. Po klesnutí segmentu pod empiricky zvolenou mez, nebo při degradaci algoritmu a překročení hloubky rekurze, přechází algoritmus do poslední čtvrté fáze.

V poslední čtvrté fázi probíhá řazení krátkých posloupností algoritmem Heap-Insertion sort, probraným v sekci 2.4.

3.4 Paralelní vícecestné rozdělování

Jedná se o první a stěžejní fázi algoritmu MPQsort. Implementace se dále dělí na tří podfáze, zachycené na schématu 3.3.

Následuje popis jednotlivých fází, diskuze možných implementací a jejich výběr. Při implementaci jsme naráželi na neefektivní části kódu, které bylo nutné hlouběji analyzovat a navrhnout optimálnější řešení.



Obrázek 3.3 Schéma jednotlivých fází v rámci paralelního vícecestného rozdělování. Šipky určují směr přechodu mezi fázemi.

Přípravná fáze

V rámci první fáze (přípravné) provádíme několik kroků. Volíme počet pivotů p na základě počtu jader poskytnutých k výpočtu. Hodnotu p získáme jako $p = 2^{\lfloor \log_2 c \rfloor} - 1$, kde c vyjadřuje kolik jader má algoritmus k dispozici. Počet pivotů se dá nastavit jako parametr, pokud bychom si přáli počet vytvořených segmentů zvětšit. Musí být však roven $2^k - 1$, kde k je přirozené číslo. Restrikce počtu pivotů jakožto mocnina dvou mínus jedna je dán algoritmem pro určení segmentu, do kterého element ze vstupní posloupnosti spadá. Díky jejich vhodně zvolenému počtu jsme schopni vytvořit úplný binární vyhledávací strom a minimalizovat tak počet porovnání na hloubku stromu, která je vždy $\lceil \log_2 p \rceil$. Funkce pro zjištění segmentu prvku nese název *find_element_segment_id*.

Dalším krokem první fáze je výběr a seřazení pivotů. Výběr pivotů probíhá pomocí tzv. *sampling* metody, probrané v kapitole 2.1. V tento okamžik na výběru pivota velmi záleží a chceme docílit velikostně nejpodobnějších segmentů. Počet elementů na výběr pivota se dá modifikovat pomocí parametru. Vzhledem k tomu, že se paralelní vícecestné rozdělování provádí pouze jednou, je vhodné uvažovat hodnoty parametru 30 a více.

Po získání pivotů je dalším krokem zjištění počtu elementů v každém segmentu. Vstupní posloupnost uniformně rozdělíme mezi vlákna pomocí paralelního cyklu for knihovny OpenMP se statickým plánováním. Každé vlákno si vytvoří lokální privátní pole velikosti s. Hodnotu pole inkrementuje na indexu i právě tehdy, když narazí na element, spadající do segmentu s_i. Pro určení segmentu elementu vlákna využívají funkci *find_element_segment_id*. Poté, co všechna vlákna se svým přiděleným úsekem skončí, provedeme redukci nad privátními poli do sdíleného pole. Tím získáme počty elementů v segmentech pro celou vstupní posloupnost. Pokud by vlákna pracovala přímo nad sdíleným polem, musely by být zápisy atomické, což by vedlo k neefektivitě.

Po průběžných měřeních a optimalizacích MPQsort algoritmu jsme zjistili, že je výše popsaný způsob zjišťování velikostí segmentu neefektivní. Doba strávená byla téměř rovna době rozdělovací fáze. Po analýze a implementaci jiných přístupů bylo zjištěno, že nevyužíváme efektivně cache paměť, konkrétně úroveň L3. Tato úroveň je totiž sdílená všemi vlákny a pokud data nejsou v L3, pak jsou načítána přímo z hlavní paměti. Po získání disjunktní části vstupního pole vláknem jsou tato data načtena do cache pamětí, ale přistupuje k nim jen jedno jádro. L3 je vůči hlavní paměti mnohonásobně menší a bylo by vhodnější, pokud by načtená data do L3 mohlo využít více vláken současně. V této fázi dochází pouze ke čtení dat, nikoli k jejich modifikaci. Pozměníme tedy algoritmus tak, abychom tohoto vylepšení docílili.

Při implementaci nového přístupu nebudeme rozdělovat vstupní posloupnost mezi vlákna, ale rozdělíme mezi ně pivoty. Pro jednoduchost budeme uvažovat situaci, kdy necháme algoritmus zvolit počet pivotů a neměníme hodnotu parametrem. V takovém případě dostane každé vlákno nejvýše jeden pivot, kromě posledního vlákna. Počet segmentů s je mocninou dvou a pokud tedy bude k dispozici například 8 vláken, pak je potřeba 7 pivotů. Pokud není počet vláken mocninou dvou, pak zůstane nevyužitých jader více. Každé vlákno prochází celou vstupní posloupnost a zaznamenává počet prvků menších než přidělený pivot. Vlákna posloupnost zpracovávají přibližně

stejnou rychlostí, a proto pracují v jeden čas na identických datech. Perfektně dodržují prostorovou lokalitu a data načtena jedním vláknem jsou využita i ostatními. Po skončení každé vlákno zapíše výsledek do sdíleného pole. Není třeba atomických operací, jelikož jsou indexy pivotů disjunktní a index přiděleného pivota je roven indexu do sdíleného pole. Při zápisu do sdíleného pole bude docházet k falešnému sdílení. Počet zápisů je však roven p a tento problém je zcela zanedbatelný. Tímto způsobem jsme získali konce segmentů, ze kterých jsme schopni spočítat i jejich začátky.

Díky úpravě výše jsme získali silné zrychlení vůči původní implementaci a to i navzdory nevyužití několika vláken během výpočtu. Pomocí parametru je možné pouze navýšit počet využitých pivotů, nikoli jej snížit a to z důvodu vyvažování zátěže. Pokud by byl zvolen vyšší počet pivotů, pak by na vlákno mohlo připadnout více než jeden pivot. V takovém případě rozdělíme počty pivotů mezi vlákna rovnoměrně. V případě, kdy je jednomu vláknu přiřazeno více pivotů, můžeme využít další optimalizační techniky a to pro redukci počtu porovnání. Seřazené pole pivotů bylo po částech rozděleno mezi jednotlivá vlákna. Porovnávaní prvků začíná s prvním pivotem vlevo, a pokud je element menší než pivot, můžeme rovnou inkrementovat i čítače ostatních pivotů nacházející se vpravo od aktuálního pivota. Element bude zajisté menší než hodnoty ostatních pivotů, a proto jej není třeba s ostatními pivoty porovnávat.

Po zjištění začátku a konce jednotlivých segmentů spočteme počet výsledných segmentů. I přes snahu vybrat optimální pivoty pro dosažení optimálního rozdělování se může stát, že segment nebude obsahovat žádné elementy a celkový počet uvažovaných segmentů bude proto nižší. Zároveň v přípravné fázi vytvoříme lambda funkci *element_in_segment* pro zjištění, zda element patří do poskytnutého segmentu a lambda funkci pro nalezení nezneutralizovaného segmentu *find_unprocessed_segment*. Tato lambda funkce buď skončí neúspěchem, kdy byly všechny segmenty zpracovány (není možné z nich vzít další blok) a nebo úspěchem, kdy nám vráti číslo nezneutralizovaného segmentu.

V rámci přípravné fáze je nutné kromě zjištění začátků resp. konců segmentů a definic pomocných funkcí připravit i pomocné tabulky. Tyto tabulky se v případě nutnosti naplní v *rozdělovací* fázi a poté se jejich obsažená data využijí v *úklidové* fázi. Budou zapotřebí dvě tabulky a dvě pole. Tabulky jsou reprezentované pomocí jednorozměrného pole se kterým pracujeme jako s dvojrozměrným z důvodu efektivity a ušetření zbytečné nepřímé indexace. Tabulky vyžadované implementací nesou postupně název *elements_table_insertions* a *elements_table*. Názvy dvou polí jsou *elements_table_insertions_index* a *elements_table_index*, které se vážou k příslušným tabulkám. Pro zkrácení značení a přehlednost v textu, budeme struktury značit pomocí zkratek *ETI* s příslušným pomocným polem *ETIi* a *ET* s pomocným polem *ETi*.

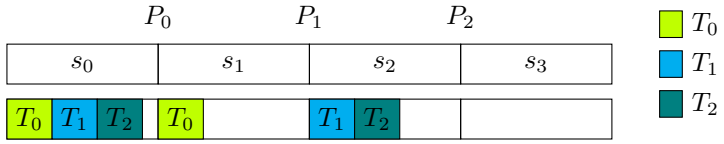
Pomocné struktury byly zapotřebí z několika důvodů, které uvedeme při popisu další fáze rozdělování. Prozatím uvedeme hodnoty, které jednotlivé struktury ukládají. První tabulka ET udržuje elementy spadající do konkrétního segmentu s_i. Pokud bude například element spadat do segmentu s_i, pak jej uložíme do řádku s indexem i. Zároveň inkrementujeme hodnotu na indexu i v poli ETi, které udržuje počet elementů pro každý řádek resp. segment. Naopak tabulka ETI ukládá hodnoty indexů elementů, nikoli elementy samotné. Index elementu je uložen do řádku i právě tehdy, když element pochází ze segmentu s_i. V poli ETIi udržujeme počet indexů obdobným způsobem, jako v poli ETi počet elementů.

Struktury popsané výše nám v úklidové fázi pomohou umístit elementy na správná místa. Po skočení druhé fáze (rozdělování) si budou hodnoty na i-tém indexu polí ETIi a ETi rovny (počet elementů odpovídá počtu segmentů). Díky těmto strukturám si můžeme udržovat přehled o tom, jaké elementy je třeba do posloupnosti vložit a také víme, na přesně jakou pozici v posloupnosti mohou být vloženy. Žádná z uvedených struktur není závislá na velikosti vstupu n a jediné vstupní hodnoty, které jejich velikost mohou ovlivnit je velikost bloku B, počet jader c využitých k výpočtu a počet segmentů s resp. počet pivotů p. Šířka tabulek ET a ETI je $(c-1)B$ a jejich výška je rovna počtu segmentů s. Pokud není parametr počtu pivotů změněn, pak bude $s \leq c$. Díky tomu i nadále dodržujeme velmi pozitivní in-place vlastnost algoritmu.

Rozdělovací fáze

Nyní si po uvedení přípravné fáze a využitých dodatečných struktur vysvětlíme hlavní část algoritmu, kterou je paralelní *rozdělování*, označované jako druhá fáze. Před vstupem do této fáze nalezneme nezpracovaný výchozí segment (tj. mohou z něj být přiřazeny bloky) pro všechna vlákna pomocí funkce *find_unprocessed_segment*. Struktury uvedené v předchozím textu budou sdíleny všemi vlákny s tím, že si každé vlákno v privátní proměnné udržuje aktuální index (ID) zpracovávaného segmentu (na počátku mají všechna vlákna společnou hodnotu) a počet zbývajících segmentů, které nebyly z pohledu daného vlákna zneutralizovány. Po vstupu do paralelního regionu druhé fáze si každé vlákno vytvoří další dvě pomocné privátní pole indexů. Velikost polí opět odpovídá počtu segmentů s. Pokud tedy budeme chtít zjistit začátek a konec bloku, který pochází ze segmentu s indexem i, pak tyto hodnoty nalezneme pod stejným indexem v příslušných polích. Díky analýze dvoucestného paralelního Quicksort algoritmu totiž víme, že je z důvodu efektivity nutné, abychom vláknům ze segmentů nepřidělovali jednotlivé elementy, ale celé bloky. Tím zlepšíme přistup ke cache pamětím a předejdeme problému falešného sdílení. Posledním krokem vláken, před zahájením vlastního paralelního rozdělování, je vytvoření lambda funkce *find_unprocessed_block*, která nalezne nezpracovaný blok pro dané vlákno, obdobně jako funkce *find_unprocessed_segment* hledá nezpracovaný segment.

V tuto chvíli mají všechna vlákna připraveno vše potřebné a mohou zahájit samotné paralelní vícecestné rozdělování. Ze začátku mají všechna vlákna nastavena segment se stejným ID. Na obrázku 3.4 si můžeme prohlédnou možné přiřazení bloků mezi vlákna s výchozím $ID = 0$. Pořadí přidělení bloků z tohoto segmentu vláknům je náhodné a záleží na rychlosti jednotlivých vláken. Vlákno atomicky získá původní hodnotu začátku segmentu a atomicky inkrementuje začátek segmentu o velikost bloku B. K této operaci slouží konstrukt OpenMP *atomic capture*. Následně na tuto hodnotu nastaví prvek pole *block_start* na indexu ID. Hodnotu pole *block_end* na indexu ID nastaví na hodnotu začátku bloku zvětšenou o B. Následně je třeba předejít případům, kdy velikost segmentu není dělitelná velikostí bloku B, a nebo pokud v segmentu již nejsou volné žádné bloky. Proto výsledný konec bloku nastavíme jako $min(block\_end[ID], segment\_end[ID])$. Následně otestujeme začátek a konec bloku. Jestliže je začátek větší nebo roven konci, pak segment již neobsahoval žádné bloky k přidělení. Snížíme počet nezpracovaných segmentů pro dané vlákno a označíme segment jako zpracovaný. Následně vlákno nalezne jiný nezpracovaný blok nebo segment a začne jej zpracovávat.



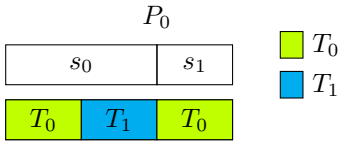
Obrázek 3.4 Schéma rozdělení vstupní posloupnosti na segmenty a přiřazení bloků vláknům z těchto segmentů.

Po získání bloku vlákny se provádí smyčka, která prochází jednotlivé elementy bloku a skončí právě tehdy, když narazí na element nepatřící do segmentu, a nebo pokud byl celý blok neutralizován. Pokud byl blok neutralizován, pak si vlákno zažádá o blok nový, který je ze stejného segmentu. Pokud segment již nemá bloky k přidělení, pak je tento segment pro dané vlákno označen jako zpracovaný a je třeba najít jiný rozpracovaný blok daného vlákna. Pokud vlákno nemá žádné rozpracované bloky, pak se pokusí vzít blok ze segmentu, který není zpracovaný. Pokud ani takový segment neexistuje, vlákno končí. Naopak pokud vlákno narazí na element nepatřící do aktuálně zpracovávaného segmentu, je třeba pomocí funkce *find_element_segment_id* zjistit ID segmentu, kam element patří. Následně vlákno element do nalezeného segmentu umístí. Aktuální segment nastavíme na nově zjištěné ID a postup popsaný výše opakujeme.

Schéma 3.4 dále ukazuje, že vlákno T_0 v bloku ze segmentu s_0 narazilo na element patřící do

segmentu s_1. Zažádalo si tedy o blok z nového segmentu, aby mělo element kam vložit. Element ze segmentu s_0 si vlákno uložilo do lokální proměnné. V této situaci nastává problém, který je potřeba vyřešit. Prvek ze segmentu s_0 patří do s_1, ale získaný prvek z druhého segmentu může patřit do jiného, než prvního. Z toho důvodu se prvky nedají napřímo prohazovat pomocí indexů do vstupního pole, ale lze je prohazovat s elementem v lokální proměnné. Blok z s_0 se označí jako *věřitel*, což znamená, že jeho element nemohl být s žádným prohozen, a proto je mu element dlužen. Díky tomuto příznaku mohou být elementy z různých bloků vždy prohazovány s elementem v lokální proměnné do té doby, dokud nenalezneme element pro blok označený jako *věřitel*. Takto označený blok vypůjčil svůj element a je potřeba tento dluh splatit. Proto element v lokální proměnné do bloku pouze vložíme a zrušíme příznak věřitele. Dále vlákno pokračuje smyčkou procházející elementy bloku, jak bylo popsáno výše. Příznak nám umožnil zachovat in-place vlastnost algoritmu.

Takto popsaný algoritmus však není korektní. Schéma 3.5 popisuje problémovou situaci. Uvažujeme $s = 2$, počet jader $c = 2$ a výchozí ID segmentu 0. Velikost segmentů je nevyvážená a platí pro ni vztah $s_0 = 2s_1$. V této situaci získalo první blok segmentu s_0 vlákno T_0, kde se nacházel element z s_1, a proto zažádalo i o blok z tohoto segmentu. Blok z prvního segmentu však obsahoval pouze jeden element z s_1 a všechny ostatní jsou na svém místě. Poté získalo vlákno T_1 blok z s_0, kde naopak všechny elementy až na jeden patří do s_1. Segment s_1 však neobsahuje žádné volné bloky. Jediný volný blok byl přiřazen předchozímu vláknu, i když do něj již žádné elementy vkládat nebude. Potřebné elementy jsou vlastněny blokem vlákna T_1 a naopak. Uvažovali jsme řešení problému pomocí předávání rozpracovaných bloků mezi vlákny dle potřeby. V takto jednoduché situaci by se mohlo jednat o možné efektivní řešení, ale s přibývajícím počtem vláken a segmentů by bylo řešení mnohem složitější. Navíc bychom se neobešli bez synchronizace mezi vlákny a museli bychom řešit jejich vzájemnou komunikaci. To by přineslo nezanedbatelné zpomalení. Z tohoto důvodu jsou pro řešení problémy využity zmiňované sdílené tabulky ET, ETI a jejich pomocná pole. Díky tomu si nemusí vlákna předávat bloky či vyměňovat informace během rozdělování. Vlákno T_1 si označí segment s_1 jako zpracovaný. V případě, kdy narazí na elementy patřící do zpracovaného segmentu, je místo umístění do bloku z cílového segmentu vloží do tabulky ET. Následně přidá i index elementu do ETI. Pro vložení je potřeba opět pouze konstrukt *atomic capture*. Pokud bychom situaci zobecnili na c vláken, pak by v nejvíce extrémní situaci mohlo nastat, že $c-1$ vláken drží všechny bloky ze segmentu i, ale elementy patřící do s_i jsou v ostatních blocích přidělovaných pouze vláknu T_0. Z této skutečnosti vyplývá šířka tabulek elementů a indexů $(c-1)B$.



Obrázek 3.5 Problémová situace, kdy blok vlákna T_1 obsahuje prvky patřící do segmentu s_1, ale všechny bloky segmentu jsou vlastněny jinými vlákny.

Uvádíme zde i pseudokód 3.1 druhé fáze. Kód je velmi zjednodušený s vynecháním implementačních detailů. Tyto detaily zahrnují pomocné proměnné, konstrukty knihovny OpenMP, vstupní a výstupní proměnné. V implementaci se navíc nejedná o separátní funkci, ale zde tuto část kódu z důvodu přehlednosti v textu jako funkci uvádíme. Části algoritmu, které jsou delší a ne přímo stěžejní pro jeho pochopení, jsou nahrazeny komentáři místo kódu.

Výpis kódu 3.1 Pseudokód druhé fáze paralelního vícecestného rozdělování.

```
ParallelMultiwayPartition()
    current_segment = default_segment_index
    while (segments_left > 0)
        if (segment[current_segment] is neutralized)
            if (saved_element exists)
                // Insert in ET and increment ETi
                if (saved_element from segment in dept)
                    // Insert element index in ETI and increment ETIi
                    block_start[segment_in_dept] += 1
                    segment_in_dept = not
                // Reset saved_element index
            if (not find_unprocessed_block(current_segment))
                if (not find_unprocessed_segment(current_segment))
                    segments_left = 0
                    continue
        else if (block_start[current_segment] >= block_end[current_segment])
            // Get block from segment with ID as current_segment
            // Set block_start and block_end
            // Shrink block_end if greater than segment boundary
            if (block_start[current_segment] >= block_end[current_segment])
                segments_left -= 1
                segment[current_segment] = neutralized
                continue
        if (current_segment == segment_in_dept)
            // Insert saved_element and reset saved index and segment
            block_start[current_segment] += 1
            segment_in_dept = not
        while (block_start[current_segment] < block_end[current_segment]
               && element_in_segment(current_element, current_segment))
               block_start[current_segment] += 1
        if (block_start[current_segment] >= block_end[current_segment])
            continue
        if (segment_in_dept == not)
            segment_in_dept = current_segment
            // Save current element, its index and segment
        else
            // Swap saved_element with current element in a current block
            block_start[current_segment] += 1
        current_segment = find_element_segment_id(saved_element)
```

Úklidová fáze

Tímto končí druhá fáze a její paralelní region. V poslední třetí fází (úklidové) je potřeba elementy z tabulky ET vložit na správné místo díky indexům z tabulky ETI. Počet elementů v řádku resp. segmentu i tabulky ET bude odpovídat počtu indexů v řádku resp. segmentu i tabulky ETI. Vkládání provádíme pomocí paralelního cyklu for a každé vlákno zpracuje i-tý segment.

Provedli jsme paralelní vícecestné rozdělování a vytvořili nejvýše s segmentů (některé nemusely obsahovat žádné elementy). Způsob zpracování vytvořených segmentů byl již popsán.

3.5 API algoritmu MPQsort

Při návrhu API algoritmu MPQsort jsme se snažili o co nejjednodušší použití z hlediska programátora a zároveň, aby bylo co nejpodobnější ostatním knihovním řadícím algoritmům jazyka C++.

Hlavní implementace se nachází ve jmenném prostoru (namespace) *mpqsort* a název funkce řazení je intuitivně *sort*. Možné způsoby volání uvádíme v 3.2. Jedná se o *header-only* knihovnu a pro její využití ji stačí pomocí *include* vložit mezi hlavičkové soubory. Způsob výběru algoritmu (sekvenční, paralelní, ...) se provádí pomocí tzv. *execution policies*. Předdefinované politiky se nachází v namespace *mpqsort::execution* a jsou taktéž uvedeny v 3.2. Uživatel má možnost poskytnout vlastní porovnávací funkci a v případě zvolení paralelní politiky i počet jader pro výpočet.

V neposlední řadě má programátor prostor pro změnu několika parametrů, které mohou ovlivnit dobu výpočtu a dají se nastavit tak, aby na zvoleném systému algoritmus běžel co nejrychleji. Z důvodu dlouhých názvů parametrů je zde nebudeme všechny vyjmenovávat, ale jen je slovně popíšeme. Všechny parametry se nachází ve jmenném prostoru *mpqsort::parameters*. Díky parametrům je možné nastavit velikost bloku, který je vláknům přidělován v druhé fázi (paralelní vícecestné rozdělování). Můžeme také nastavit meze, kdy po klesnutí délky vstupní posloupnosti řadíme data pouze vícecestným sekvenčním algoritmem a v případě velmi krátkých posloupností dojde k seřazení Heap-Insertion algoritmem. Dále je možné zvolit počet pivotů určených k vícecestnému paralelnímu rozdělování. Tím zároveň nastavíme počet výsledných segmentů. Na závěr jsme schopni nastavit počet elementů pro výběr jednoho pivota v paralelním vícecestném rozdělování a také jejich počet v paralelním vícecestném Quicksort algoritmu.

Výpis kódu 3.2 Rozhraní API řadícího algoritmu MPQsort a jeho možné volání.

```
#include <mpqsort.h>
#include <vector>

using namespace mpqsort;

int main() {
    std::vector<int> v{2, 3, 5, 3};
    auto cmp = [](int a, int b) { return a > b; };
    int cores = 8;

    // Possible MPQsort overloads for two-way sequential qsort
    sort(v.begin(), v.end());
    sort(v.begin(), v.end(), cmp);
    // Possible MPQsort overloads for three-way and four-way sequential qsort
    sort(execution::seq_three_way, v.begin(), v.end());
    sort(execution::seq_three_way, v.begin(), v.end(), cmp);
    sort(execution::seq_four_way, v.begin(), v.end());
    sort(execution::seq_four_way, v.begin(), v.end(), cmp);
    //Possible MPQsort overloads for parallel multi-way qsort
    sort(execution::par, v.begin(), v.end());
    sort(execution::par, cores, v.begin(), v.end());
    sort(execution::par, v.begin(), v.end(), cmp);
    sort(execution::par, cores, v.begin(), v.end(), cmp);
}
```

Kapitola 4
Testování implementace

Nedílnou součástí vývoje a implementace paralelního vícecestného quicksort algoritmu *MPQsort* je jeho důkladné otestování. Testování zahrnuje jak korektnost implementace pomocí unit testů, tak důkladné naměření a vyhodnocení jeho efektivnosti.

Pro otestování korektnosti byla využita knihovna *Catch2* a pro měření efektivnosti řazení knihovna *Google benchmark*, což usnadnilo, zpřehlednilo a dopomohlo k vyhodnocení dosažených výsledků implementace. Obě z uvedených knihoven jsou často využívány v jazyku C++.

Google benchmark umožňuje provádět měření částí kódu podobným způsobem, jako testování korektnosti implementace pomocí unit testů. [18] Psaní benchmarků je obdobné jako psaní unit testů v kapitole 4.2 s využitím knihovny *Catch2*. Google benchmark umožňuje využití tzv. *fixtures*, což jsou pomocné třídy sloužící k přípravě testovacího prostředí v rámci testu. V našem případě chceme otestovat vícero řadících algoritmů s identickými vstupními daty. Fixtures nám umožní implementovat generování dat pouze jednou a vygenerovat data znovu při každém volání benchmark funkce. Další výhodou knihovny je psaní parametrizovaných testů, což například umožňuje implementovat test pouze jednou a následně jej spouštět pro různě veliké vstupy či datové typy. Součástí knihovny je také nástroj na porovnávání naměřených dat mezi sebou a vytváření tabulek, což během implementace umožnilo přehledně a jasně odladit implementaci za cílem vyšší efektivity. Knihovna nabízí široké možnosti a nástroje. My jsme si shrnuli jen ty nejdůležitější funkcionality pro měření efektivity implementace *MPQsort* a porovnání s ostatními.

V této kapitole si nejdříve ve stručnosti probereme testovací prostředí clusteru STAR a výše zmíněné knihovny využité pro testování korektnosti a efektivity výsledné implementace. Poté popíšeme přípravu a generování testovacích dat, včetně jejich vlastností z hlediska uspořádání.

Následuje kapitola vyhodnocení a měření efektivity pro různé hodnoty parametrů algoritmu *MPQsort* a jejich vliv na rychlost řazení. V kapitole 5 probíhá vyhodnocení rychlosti algoritmu pro různé datové typy, uspořádání vstupních dat, mohutnost množiny oboru hodnot a škálování se zvětšujícím se počtem jader.

S řádně naměřenou a otestovanou efektivitou algoritmu *MPQsort* následuje porovnání s existujícími implementacemi paralelních řadících algoritmů pro jazyk C++. Porovnání bude probíhat vůči standardním a rozšířeným paralelním řadícím algoritmům od GNU a dále s algoritmy AQsort [17] a cpp11sort [9].

4.1 Testovací prostředí

Všechna měření byla prováděna na clusteru STAR. Uživatel se po připojení ke clusteru nachází na front-end uzlu, kde může kompilovat svůj algoritmus a přidávat úlohy do výpočetní fronty.

Přímé spuštění na back-end uzlech není možné a je potřeba využít plánovač, který úlohu

do fronty vloží. Úlohy jsou plánovačem odebírány a spouštěny takovým způsobem, aby nedošlo k ovlivnění měření jinými úlohami. Na jednom back-end uzlu by tedy vždy měla být spuštěna pouze jedna úloha, čímž jsme schopni získat neovlivněné výsledky.

Výpočetní back-end uzly jsou osazeny procesorem Intel Xeon E5-2630 v4 2.20GHz s celkem dvaceti jádry. Základní deska osazena tímto procesorem totiž obsahuje dva sockety a v každém socketu je jeden procesor. Jedná se tedy o architekturu NUMA a každý procesor má deset jader. Procesor sice podporuje technologii hyper-threading, ale pro přesnější měření je na uzlu vypnuta. Hlavní paměť má více než dostatečnou velikost pro naše potřeby a to 64 GB. Velikosti jednotlivých úrovní cache L1, L2 a L3 jsou 32 KB, 256 KB a 25600 KB.

V době měření měl STAR systém CentOS ve verzi 7.9.2009 (Core) 64bit a verzi překladače GCC 10.2.1, který byl využit pro kompilaci všech komponent a algoritmů. Verze standardu C++ byla při kompilaci nastavena na C++ 17.

4.2 Korektnost implementace

Pro ověření korektnosti implementace řadícího algoritmu, pomocných metod a dílčích kroků jsme vybrali testování pomocí *unit testů*. V našem případě se jedná o nejlepší způsob jak testovat korektnost jednotlivých kroků algoritmu. Jednotlivé části kódu jsou rozděleny do jednoduchých pomocných funkcí, a proto mohou být testovány nezávisle na sobě. Jedná se o nejmenší možnou testovatelnou jednotku programu a to nám umožní podchytit chyby včas a zjednoduší jejich lokalizaci.

V rámci testování budeme také využívat *AddressSanitizer*. [19] AddressSanitizer nám pomůže detekovat nekorektní práci s pamětí jako je přístup mimo hranice pole, dereference *nullptr* ukazatele a v neposlední řadě úniky paměti. *ThreadSanitizer* zvládne detekovat současný přístup vláken ke stejné proměnné v případě současného zápisu vícero vláken, nebo současného čtení a zápisu vícero vláken. V případě, že tato situace nastane, nás na ni upozorní. Oba nástroje se spouští pouze přidáním parametru v době kompilace, což může běh programu zpomalit až 20x. Z toho důvodu bude kód kompilován s těmito přepínači jen v případě testování korektnosti a ne efektivnosti.

Při testování jsme narazili na chybu v knihovně *Catch2* ve spojení s detekcí ThreadSanitizer a vícevláknovou aplikací. Kvůli této chybě získáváme množství chyb, které však nejsou způsobeny nekorektní implementací vícevláknového algoritmu a jsou tedy falešně pozitivní. Kvůli tomuto problému byla kontrola odebrána a při testování nebyla uvažována.

Pro unit testy v C++ existuje řada knihoven, z nichž nejznámější jsou *Google Test*, *Boost.Test*, *doctest* a *Catch2*. Všechny uvedené knihovny poskytují kvalitní testovací framework, a proto bylo poměrně složité vybrat tu nejlepší pro naše účely. Naše požadavky na testovací framework jsou následující:

Assert makra Testovací framework musí obsahovat předdefinovaná assert makra sloužící na ověření pravdivosti výrazu. Dále by měla umožňovat přerušení zbývajících testů v případě neúspěchu, ale také jejich pokračování.

Generátory Implementujeme řadící algoritmus, a proto by bylo vhodné mít možnost generovat data jako vstup do algoritmu. Data by mělo být možné generovat náhodně a nebo jako pole s definovanými vlastnostmi.

Vytváření skupin Umožnit jednotlivé testy seskupovat do větších celků a mít možnost spustit pouze definovanou množinu testů.

Fixtures Často se stává, že různé testy vyžadují stejné prostředí (data, komponentu, ...). V našem případě můžeme mít testy, které testují různé parametry řadícího algoritmu, ale všechny vyžadují data co mají seřadit. Příprava dat může být implementována jednou pro všechny tyto testy.

Jednoduchost a přehlednost Požadujeme co nejjednodušší použití a instalaci testovacího frameworku, ale ne na úkor funkcionalit výše. Také požadujeme přehlednost výsledků a možnost výpisu proměnných.

Google Test je pravděpodobně nejrozsáhlejší testovací framework z dříve vyjmenovaných. Zahrnuje velké množství předdefinovaných *assert* maker, poskytuje možnost *mocks* (dokáže simulovat chování jiného objektu). Má však složitější konfiguraci, je potřeba stáhnout a nalinkovat knihovnu. Poskytuje fixtures, ale nemá implementované generátory. [20]

Boost.Test je součástí knihovny *Boost* a poskytuje assert makra s možností C++ logických operátorů, obsahuje generátory, seskupování testů a dokáže implementovat fixtures. Je jednodušší na instalaci, ale z mého pohledu nejsou testy tolik přehledné. [21]

Doctest framework implementace je header-only, tudíž stačí stáhnout jeden soubor, přidat ho mezi include a vše je připraveno. Není nutné kompilovat a linkovat externí knihovny. Testy mohou běžet paralelně a oproti Catch2 se testy rychleji kompilují a provádějí. Neposkytuje implementaci generátorů. [22]

Catch2 je posledním z vyjmenovaných frameworků. Stejně jako *doctest* poskytuje implementaci header-only, jeho využití je jednoduché a přehledné. Implementuje generátory a také tzv. *matchers*, což jsou speciální funkce na vyhodnocení složitějších podmínek (porovnání dvou vektorů, texty výjimky, porovnání čísel s plovoucí desetinnou čárkou, ...). Z hlediska funkcionalit disponuje i speciálními makry pro testování template tříd, funkcí a struktur jazyka C++. Díky tomu je možné spustit identický test pro vícero typů velice jednoduše. Výpis testů je přehledný a dostatečně expresivní pro následný debugging. V neposlední řadě disponuje i možností *micro benchmarking*, což umožňuje měření času vykonávání vybraných funkcí či fragmentu kódu. Pro tato měření jsem se však rozhodl využít *Google benchmark* knihovnu, která nabízí rozsáhlejší možnosti měření. [23]

Na základě možností různých testovacích knihoven jsem se rozhodl využít Catch2. Splňuje všechny mnou kladené požadavky na testovací framework, je jednoduchá na použití a disponuje přehlednou dokumentací. Příklady definice testů a jejich spouštění si stručně ukážeme v následující sekci.

Spouštění testů a jejich definice

Pro implementaci testů je potřeba přidat header-only knihovnu *Catch2* do námi implementovaného programu, přidáním hlavičkového souboru *catch2.hpp*. Implementace testů se dělí do dvou souborů. Testy jako takové jsou umístěny v souboru *test/source/mpqsort.cpp* a soubor *test/source/main.cpp* slouží na jejich spouštění a implementuje funkci main.

Výpis kódu 4.1 Ukázka souboru implementujícího funkci main pro spouštění testů

```
1 // File main.cpp
2 #define CATCH_CONFIG_MAIN
3 #include <catch2/catch.hpp>
```

Výpis kódu 4.1 předvádí jednoduchost implementace funkce main pro spouštění testů. Makro preprocesoru CATCH_CONFIG_MAIN automaticky vygeneruje definici main včetně přepínačů. Pokud bychom chtěli, máme možnost funkci main implementovat sami a upravit její možnosti a chování. To však v našem případě nebude potřeba, jelikož originální varianta poskytuje vše, co budeme pro testování řadících algoritmů potřebovat.

Na výpisu 4.2 si můžeme prohlédnou jeden z příkladů testování korektnosti řazení implementovaného algoritmu **mpqsort::sort**. Makro GENERATE knihovny Catch2 způsobí to, že test bude spouštěn opakovaně a proměnná *size* bude postupně nabývat hodnot $(1, 2, 4, 6)$. Na základně toho upravíme velikost vektoru. Díky tomu můžeme vytvořit jeden test, který přitom testuje vícero velikostí vstupu. Tím ušetříme velké množství kódu a umožní nám definovat testy

rychleji a přehledněji. Následující makro SECTION způsobí podmíněné vykonávání kódu v těle testu. Obdobně jako u makra GENERATE se test spustí opakovaně a v každém spuštění se vykoná pouze jedna sekce. Obě sekce předpokládají, že bude připraven vektor s daty, která se mají seřadit. Kód pro takovou přípravu bude pro všechny sekce společný. Následně každá sekce volá řadící algoritmy s jinými parametry a očekává jiné výsledky. Díky tomu můžeme zkombinovat vícero testů dohromady a opět zpřehlednit a zkrátit výsledný kód. Korektnost výsledku je v tomto případě zkontrolována pomocí makra REQUIRE_THAT a funkce Catch::Equals, která je jednou z předdefinovaných *matcher* funkcí. Tento zápis na jeden řádek porovná identičnost dvou vektorů a v případě nerovnosti nahlásí chybu, vypíše ji a následující testy nespouští. Pokud bychom chtěli v testech pokračovat i v případě chyby a jen ji vypsat, můžeme využít makro CHECK_THAT.

Framework Catch2 nabízí širší množství typů testů, maker, generátorů a pomocných funkcí, které v testování využívám. Výpis 4.2 slouží jako demonstrace a důvod, proč jsem si tento testovací framework vybral.

Spuštění testů je jednoduché, jelikož po kompilaci se nám vytvoří spustitelný soubor po jehož spuštění se provedou námi definované testy. Při spouštění můžeme využít dodatečné argumenty a parametry. Tím jsme schopni ovlivnit, které testy se spustí, jaký formát výpisu požadujeme (json, csv, cmd) a jiné užitečné možnosti.

Výpis kódu 4.2 Ukázka testování kódu pomocí Catch2

```
#include <catch2/catch.hpp>
#include <vector>
// Test if mpqsort:sort returns correct results. Expected result is computed
↪  using std::sort
TEST_CASE("Sorting vector of different sizes and custom comparators") {
    // Runs once per generator value
    auto size = GENERATE(1, 2, 4, 6);
    std::vector<int> test_vector{5, 3, 3, 6, 6, 7};
    auto first = test_vector.begin();
    auto last = test_vector.end();
    auto res = test_vector;
    // Resize based on size value. For each test run is size different
    test_vector.resize(size);
    SECTION("Default comparator") {
        mpqsort::sort(first, last);
        std::sort(res.begin(), res.end());
        REQUIRE_THAT(test_vector, Catch::Equals(res));
    }
    SECTION("Greater comparator") {
        mpqsort::sort(first, last, std::greater<int>());
        std::sort(res.begin(), res.end(), std::greater<int>());
        REQUIRE_THAT(test_vector, Catch::Equals(res));
    }
}
```

4.3 Příprava testovacích dat

Nejdůležitějším požadavkem na testovací data u variant Quicksort algoritmu jsou identická vstupní pole při opakovaných měřeních. Na rozdíl od datově necitlivých algoritmů (Mergesort, Heapsort, ...) je Quicksort a jeho varianty datově citlivý a i v případě permutací identického

vstupu bychom mohli dostat velmi odlišné výsledky. Při testování je také důležité mít dostatečně veliké vstupy, jelikož jsou řadící algoritmy zpravidla rychlé (obzvláště paralelní verze). Velikost také pomůže k využití všech jader daného CPU a naměřené výsledky budou přesnější. Délka posloupnosti, splňující tyto požadavky, je od 8 GB dat a více.

Vlastnosti dat

Vlastnosti testovacích dat se lišily v datovými typy, vstupním uspořádáním a mohutností oboru hodnot. V rámci měření byly uvažovány tři základní datové typy, které se běžně využívají v praxi:

- **int** - celá čísla (velikosti 4 B)
- **short** - celá čísla poloviční délky (velikosti 2 B)
- **double** - desetinná čísla s dvojitou přesností dle IEEE 754

Interface *MPQsort* a ostatních řadících algoritmů umožňuje uživateli poskytnout porovnávací funkci a seřadit libovolné datové typy. Ostatní datové typy jsme však vynechali, jelikož základní datové typy považujeme za dostatečné a jsou také běžně využívány při prezentaci výsledků řadících algoritmů.

Velikost vstupních dat je vždy určena s ohledem na to, aby se data vešla do paměti. Řadící algoritmus si může a bude alokovat dodatečnou paměť i v případě in-place verze (záloha registrů na stack v případě rekurze, pomocné struktury).

Abychom mohli korektně porovnat efektivitu mezi různými datovými typy, musí jich být identický počet. Kvůli skutečnosti, že datový typ *double* je zpravidla dvakrát větší (8 B) oproti datovému typu *int* (4 B), musí brán zřetel právě na double. Tento datový typ určuje maximální množství alokované paměti algoritmem během měření a nesmí přesáhnout paměť systému.

Z hlediska vstupního uspořádání posloupností jsou uvažovány následující typy:

- Pseudonáhodná posloupnost
- Seřazená posloupnost
- Opačně seřazená posloupnost
- Varhanově seřazená posloupnost
- Rotovaná seřazená posloupnost

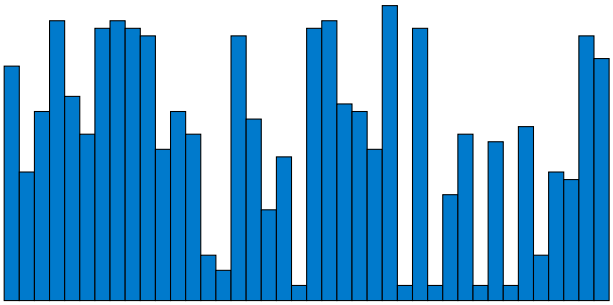
Pseudonáhodná posloupnost je generována z uniformního rozdělení pomocí *Mersenne Twister 19937* a je využita jako vstup pro ostatní typy uspořádání. Detaily ohledně generování dat jsou uvedeny v kapitole 4.3. Schéma náhodného uspořádání je pro lepší představu vyobrazeno na 4.1a.

Seřazená posloupnost se myslí ve vztahu vůči využité porovnávací funkci. Tedy pokud porovnávací funkce f vrátí hodnotu $true$ pro prvky, kdy platí $a < b$, a v ostatních případech hodnotu $false$, pak je vstupní seřazená posloupnost vzestupná. Ukázku, jak seřazená data vypadají, si můžeme prohlédnout na obrázku 4.1b.

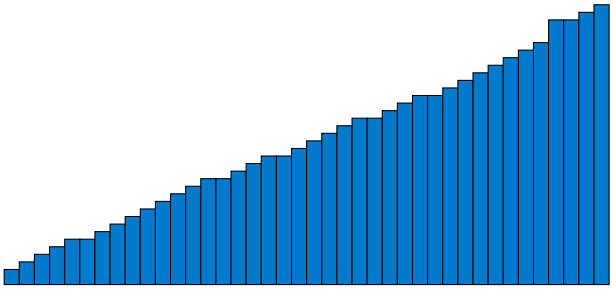
Opačně seřazená posloupnost je myšlena opět vůči porovnávací funkci. V případě využití funkce f a její negace budou vstupní data opačně seřazená vůči datům na obrázku 4.1b a výsledné uspořádání vidíme na 4.1c.

Varhanově seřazená data jsou tvořena dvěma posloupnostma. Obě mají délku $n/2$, kde n je celková délka pole. První je neklesající od indexu pole 0 až po $n/2 - 1$. Druhá je nerostoucí od indexu $n/2$ až po $n - 1$. Výsledek tvarově připomíná varhany a odtud nese svůj název 4.1d.

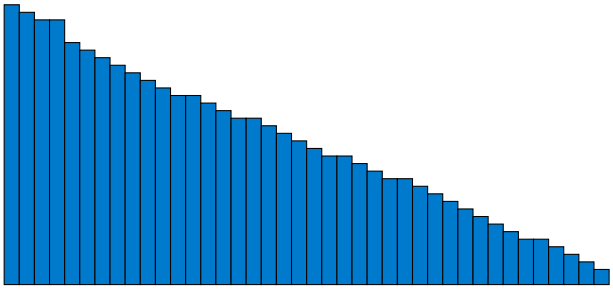
Poslední ze vstupních posloupností je tzv. rotovaná. Jedná se o seřazenou posloupnost s tím rozdílem, že jsou všechny prvky cyklicky posunuty o jednu pozici vlevo.



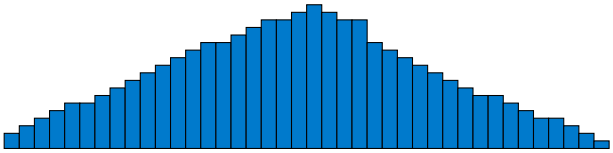
(a) Schéma náhodného seřazení čtyřiceti elementů.



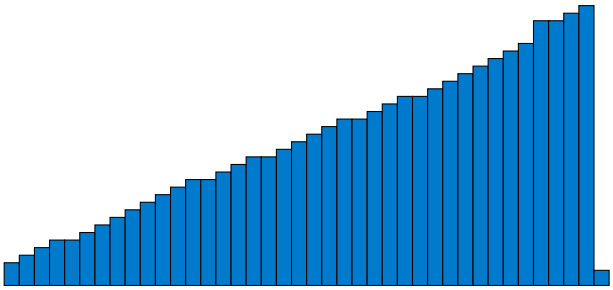
(b) Schéma seřazených čtyřiceti elementů.



(c) Schéma opačně seřazených čtyřiceti elementů.



(d) Schéma varhanově seřazených čtyřiceti elementů.



(e) Schéma seřazených čtyřiceti elementů a následně cyklicky posunutých o jednu pozici vlevo.

Obrázek 4.1 Různé typy vstupního uspořádání pro otestování implementací.

Součástí testování efektivnosti bylo otestováno chování algoritmu v případě různých mohutností oboru hodnot. V praxi se zpravidla setkáváme s posloupnostmi, kde jednotlivé prvky nabývají omezeného množství hodnot, což způsobuje opakování jednotlivých elementů. Při testech, které se nezaměřují na tuto vlastnost, mohly prvky nabývat jakékoli validní hodnoty vybraného datového typu.

Generování dat

Prvotním záměrem bylo vygenerování vstupních dat a uložení do souboru. To přinášelo několik velmi nepraktických problémů. V první řadě by vygenerovaná vstupní data zabírala nepřiměřené místo na disku, jelikož pro každý datový typ by bylo potřeba vytvořit jiný typ posloupnosti a jen tato vygenerovaná data by zabírala nejméně 120 GB (v případě velikosti 8 GB pro jeden vstup). Druhou nevýhodou byla doba generování a načítání těchto dat. Generování dat a alokace potřebné paměti je rychlá operace, úzkým hrdlem se jeví přenosová rychlost disku. Proto budeme data generovat vždy nově a přímo do paměti bez zbytečného zápisu a čtení ze souboru.

Pro generování využijeme *Mersenne Twister 19937*, což je nejrozšířenější generátor pseudonáhodných čísel knihovny STL. [24] Tento generátor přijímá jako vstupní parameter tzv. *seed*, který slouží k jeho inicializaci. Pokud je seed mezi voláními identický, pak generátor vyprodukuje vždy stejnou pseudonáhodnou posloupnost. Hodnoty posloupnosti jsou generovány z uniformního rozdělení, tedy každou hodnotu z intervalu $[a, b]$ vygenerujeme se stejnou pravděpodobností.

Generování dat je možno paralelizovat, kde každé vlákno dostane instanci generátoru *Mersenne Twister 19937* s odlišnou hodnotou *seed*, kde seed odpovídá číslu vlákna. Číslo vlákna je jeho unikátní identifikátor a nabývá hodnot $\{0, \ldots, p-1\}$, kde p je počet vláken. Díky unikátnímu seed bude každé vlákno generovat svou unikátní pseudonáhodnou podposloupnost výsledné posloupnosti. V opačném případě by každé vlákno generovalo podposloupnost identickou. Při paralelizaci for-cyklu generujícího vstupní posloupnosti je využito OpenMP a tzv. *static scheduling*. Díky tomu dostane každé vlákno p stejný seed a stejnou část vstupního pole, do kterého vygeneruje vždy stejnou podposloupnost. To zajistí vždy stejná vstupní data při opakovaném spuštění testu.

Náhodně vygenerovaná data slouží jako vstup pro další typy seřazení vstupní posloupnosti. Ve všech ostatních případech (kromě pseudonáhodné posloupnosti) bylo potřeba data efektivně a hlavně korektně seřadit. K tomu byla využita implementace paralelního balancovaného Quicksort algoritmu od GNU. V době psaní benchmark testů ještě nebyl MPQsort implementován a v průběhu jeho psaní bylo potřeba provést testování efektivnosti dílčích kroků (zpravidla pomocných funkcí). Původně testy využívaly implementaci standardní knihovny *STD* a její paralelní verzi algoritmu *sort*. V průběhu testování se však vyskytly potíže, jelikož algoritmus alokoval neúměrné množství paměti vůči velikosti problému a následně ji neuvolnil. To vedlo k zaplnění celé paměti systému a pádu aplikace. Z toho důvodu byl i vyřazen z porovnání efektivnosti vůči MPQsort v následující kapitole.

Kapitola 5
Diskuze naměřených výsledků

Efektivita byla měřena na různých uspořádáních dat, kdy jsme uvažovali náhodně vygenerovaná vstupní data, předem seřazená vstupní data, opačně seřazená data, varhanově seřazená data a seřazená data, která byla cyklicky posunuta o jednu pozici vlevo. Zkoumána byla i efektivita na různých datových typech a mohutnostech množiny oboru hodnot. Mezi uvažované celočíselné typy patří *int*, *short* a mezi uvažované typy s plovoucí řádovou čárkou typ *double*.

Optimální parametry implementace byly hledány na náhodných datech, která jsou zpravidla považována za nejnáročnější vstup pro řadící algoritmy. Po nalezení optimálních parametrů byla provedena měření a hodnoty parametrů se již nadále neměnily. Je možné, že pro specifické konfigurace vstupních dat existují i optimálnější parametry, ale naším cílem je, aby byl algoritmus efektivní obecně a ne pro specifický typ uspořádání či datový typ.

Pokud nebude uvedeno jinak, pak byla pro měření zvolena náhodná vstupní posloupnost s optimálními parametry algoritmu na maximálním počtu dostupných jader. V našem případě cluster STAR nabízí nejvýše 20 jader.

5.1 Nastavení optimálních parametrů MPQsort

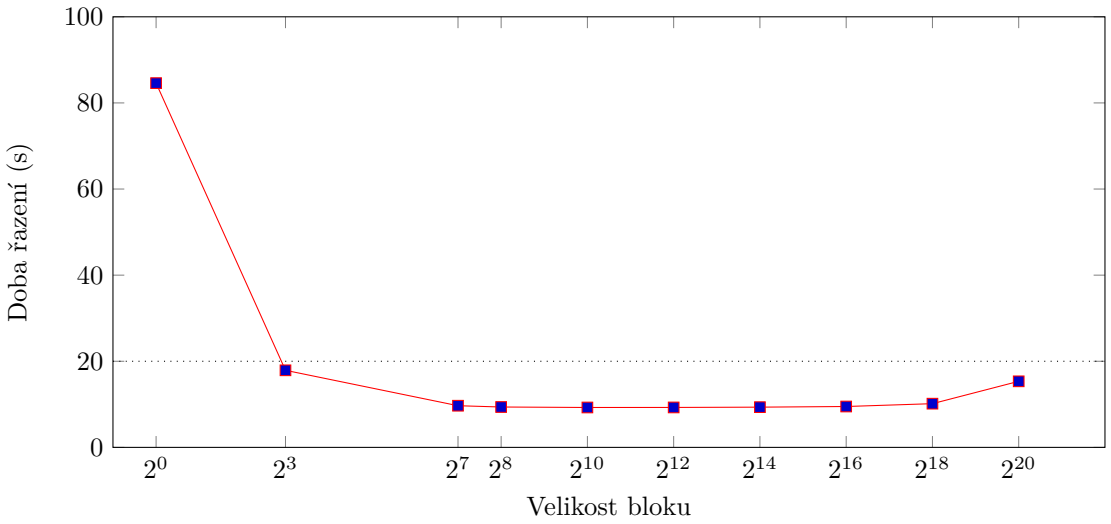
Již v sekci 3.2 jsme uvedli parametry, které se dají algoritmu MPQsort nastavit a ovlivnit tak dobu běhu. Jednotlivé parametry byly nastavovány již během implementace, když jsme rozpracovanou implementaci postupně optimalizovali a hledali zdroje neefektivity. V této sekci již pracujeme s výslednou implementací, která obsahuje dodatečné optimalizace. Cílem této sekce je najít optimální konfiguraci parametrů pro finální implementaci. S výslednou konfigurací následně provést ostatní měření a porovnání s dosavadními implementacemi. Pro získání co nejlepších výsledků volíme vstupní uspořádání jako pseudonáhodnou posloupnost. Jedná se zpravidla o nejnáročnější vstup řadících algoritmů, což nám umožní parametry optimálně nastavit.

Velikost bloku

Jedním z nejzásadnějších parametrů je právě velikost přiřazovaného bloku vláknům. Přiřazování probíhá v blocích za účelem zmírnění či naprosté eliminace falešného sdílení a omezení počtu atomických operací. Bloky jsou přiřazovány z jednotlivých segmentů, jejichž pozice byla zjištěna v přípravné fázi paralelního vícecestného rozdělování.

Malou velikostí bloků se zpravidla omezí využití pomocných struktur v případě nevyrovnanosti velikosti segmentů a velikost samotných struktur, která je na velikosti bloku závislá. Naopak způsobí vyšší počet synchronizačních konstruktů mezi vlákny (atomické operace). Veliké bloky

mají tendenci způsobit intenzivnější využití pomocných struktur a také zvětší jejich paměťovou náročnost.



Obrázek 5.1 Vliv velikosti bloku na rychlost řazení $n = 10^9$ náhodných celých čísel (int).

Provedli jsme sadu měření pro různě velikosti bloku a naměřili její vliv na rychlost běhu algoritmu. V grafu 5.1 vidíme silný nárůst pro velikost bloku 1 a její strmý pokles už při zvětšení velikosti na 2^3. Z naměřených hodnot vyšla velikost $2^{10} = 1024$ jako nejvhodnější. Rychlost algoritmu je pro bloky od velikosti 2^7 po 2^{14} velmi podobná, ale pro vybranou velikost byla o několik desítek milisekund lepší. V případě menších velikostí bloků nebo naopak větších dochází ke zpomalení algoritmu. Počet elementů 1024 je také vhodných z hlediska paměti, jelikož se od velikosti bloku B odvíjí velikost pomocných struktur a tu je vhodné také udržet minimální.

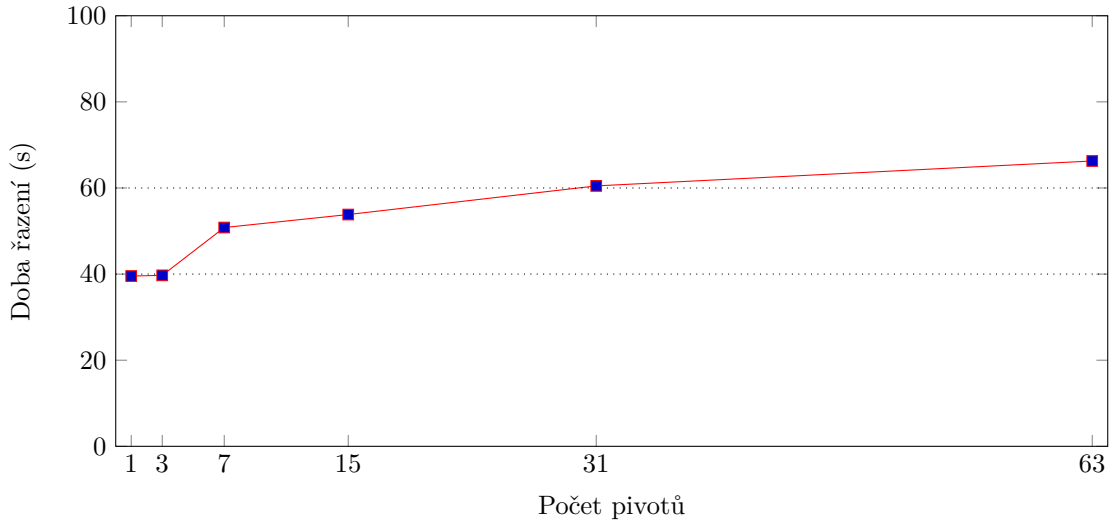
Počet pivotů

Dalším nastavitelným parametrem algoritmu *MPQsort* je počet pivotů p využitých během fáze paralelního vícecestného rozdělování 3.2. Při zvolení p pivotů získáme s segmentů a vztah mezi počtem pivotů a segmentů je $p + 1 = s$. Vzhledem k nutné přípravné fázi provádíme vícecestné rozdělování jen jednou a cílem je získat dostatečný počet segmentů, aby v dalších fázích mělo každé vlákno přiřazenou práci a nedocházelo k hladovění. Povolený počet pivotů musí odpovídat $p = 2^k - 1$, kde k je přirozené číslo.

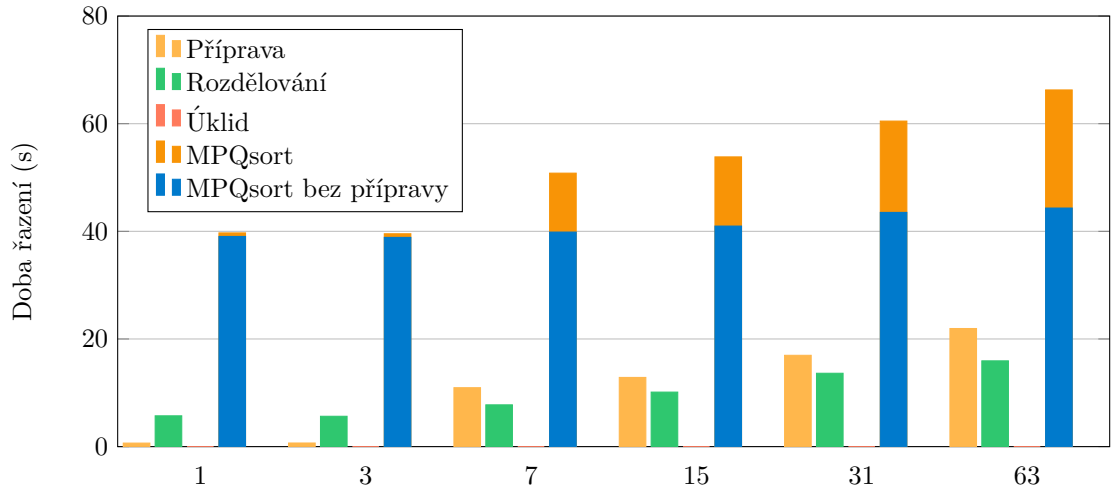
Při měření byl pro výpočet nastaven fixní počet jader 4. Důvodem je různá implementace přípravné fáze pro $p < c$, kde c je počet přidělených jader, a pro $p > c$. V prvním případě každé vlákno zpracovává jen jednoho pivota, který je uložen do registru a počet elementů menší než pivot je taktéž v registru. V druhém případě je zpracováváno vícero pivotů, které jsou čteny z vektoru a počty elementů menší než pivoti jsou také ukládány do vektoru.

Graf 5.2 zachycuje dobu řazení dat pro různý počet pivotů v případě zapojení čtyř jader. V případě kdy $p < c$ je doba řazení velmi podobná, ale při rozdělování na více segmentů ($s > c$) se řazení výrazně zpomalí. Zpomalení je nejpatrnější při přechodu ze tří pivotů na sedm. S dalším přidáváním pivotů se zpomalení pouze více prohlubuje a při rozdělení vstupu do 64 segmentů je doba řazení až 1.6 krát pomalejší než při rozdělení do čtyř segmentů. Takto výrazné zpomalení je přinejmenším podezřelé a z toho důvodu jsme se rozhodli změřit dobu strávenou v jednotlivých fázích algoritmu a přijít na hlavní příčinu.

Pro vytvoření grafu 5.3 byla použita identická konfigurace parametrů algoritmu a vstupní data, jako u 5.2. V grafu jsou zachyceny časy strávené v jednotlivých fázích vícecestného para-



Obrázek 5.2 Vliv počtu pivotů na rychlost řazení $n = 10^9$ náhodných celých čísel (int). Počet jader byl po dobu měření nastaven na fixní hodnotu $c = 4$.



Obrázek 5.3 Doba strávená v jednotlivých fázích algoritmu pro různý počet pivotů. Do posledního sloupce je vykreslena doba řazení, pokud bychom nezapočítávali dobu strávenou v přípravné fázi.

lelního rozdělování, společně s celkovou dobou běhu algoritmu *MPQsort*. V posledním sloupci je vykreslena celková doba řazení s přípravnou fází a bez přípravné fáze, za účelem lepšího porovnání jednotlivých konfigurací.

Z grafu je jasně patrné, že pro případ kdy $s < c$ je doba přípravy zcela zanedbatelná vůči celkové době řazení. Naopak v ostatních případech doba přípravy přesahuje čas strávený v samotném paralelním rozdělování. Úklidová fáze je ve všech uvažovaných případech zanedbatelná.

Skokové zpomalení je tedy způsobeno přípravnou fází a skutečností, že v případě většího počtu pivotů na vlákno je potřeba pracovat s vektory. Čítače elementů menších než pivot tedy nejsou uloženy v registru procesoru. Dalším problémem je fakt, že pro každý element nastává inkrementace vícero prvků ve vektoru čítačů. I kdybychom pro případy $p > c$ dosáhli podobné rychlosti přípravné fáze, jako u $p < c$, nedosáhli bychom s vyšším počtem pivotů zrychlení. Při porovnání modrých sloupců (MPQsort bez přípravy) se s vyšším počtem pivotů prodlužuje doba

paralelního vícecestného rozdělování.

Z analýzy naměřených hodnot v grafech 5.2 a 5.3 vychází nejlépe konfigurace, kdy $p < c$ a z důvodu vyvažování zátěže je nejvhodnějším volit počet pivotů takovým způsobem, aby počet segmentů s odpovídal počtu přiděleným jádrům. Dnešní procesory mají zpravidla počet jader roven mocnině dvou. V takovém případě volíme $p = 2^{\lfloor log_2 c \rfloor} - 1$, čímž získáme optimální počet segmentů. Cluster STAR však nabízí dvacet jader a v takovém případě algoritmus zvolí $p = 15$, čímž získáme $s = 16$. Úlohy pro zbylá jádra budou vytvořeny během další fáze 3.2, čímž případně vzniklé nevyvážení zátěže vykompenzujeme.

Výběr pivota

Výběr pivota je obzvláště důležitým parametrem v případě klasických dvoucestných Quicksort algoritmů, např. AQsort a cpp11sort. Skutečnost, že je po dobu paralelního rozdělování využit pouze jeden pivot, klade na jeho výběr silný důraz, a proto je snaha jej vybrat jako medián vstupních dat resp. skoromedián. Dalším faktorem je, že výběr pivota probíhá na každé hladině stromu rekurzivních volání, a proto v případě metody výběru *sampling* může vysoký počet sampling elementů algoritmus zpomalit.

V případě námi navrženého algoritmu *MPQsort* probíhá výběr pivotů pro paralelní vícecestné rozdělování pouze jednou, a z toho důvodu si můžeme dovolit výběrem strávit vice času. V diplomové práci Kláry Schovánkové [9] byl počet elementů pro výběr pivota nastaven na hodnotu 30. Ze vstupních dat se vybere třicet elementů, ty se následně seřadí a medián se volí jako pivot. Tento postup je opakován při každém rekurzivním volání paralelního Quicksort algoritmu.

V našem algoritmu využíváme více způsobů volby pivota a to v závislosti na fázi 3.2, ve které se algoritmus nachází. V případě paralelního vícecestného rozdělování provádíme výběr pivota metodou sampling a parametr počtu elementů na výběr jednoho pivota má hodnotu 100 (zdůvodnění dále v textu). To znamená, že při výběru p pivotů provádíme výběr z $100p$ elementů vstupních dat. V případě narůstajícího počtu jader pro výpočet algoritmus ve výchozím nastavení zároveň zpřesňuje výběr pivotů, jelikož je počet pivotů na počtu jader závislý (za předpokladu identických velikostí vstupu). Právě z důvodu většího množství pivotů neměl tento parametr až tak markantní vliv na rychlost algoritmu jako v případě dvoucestných variant a doba řazení se lišila v rámci stovek milisekund.

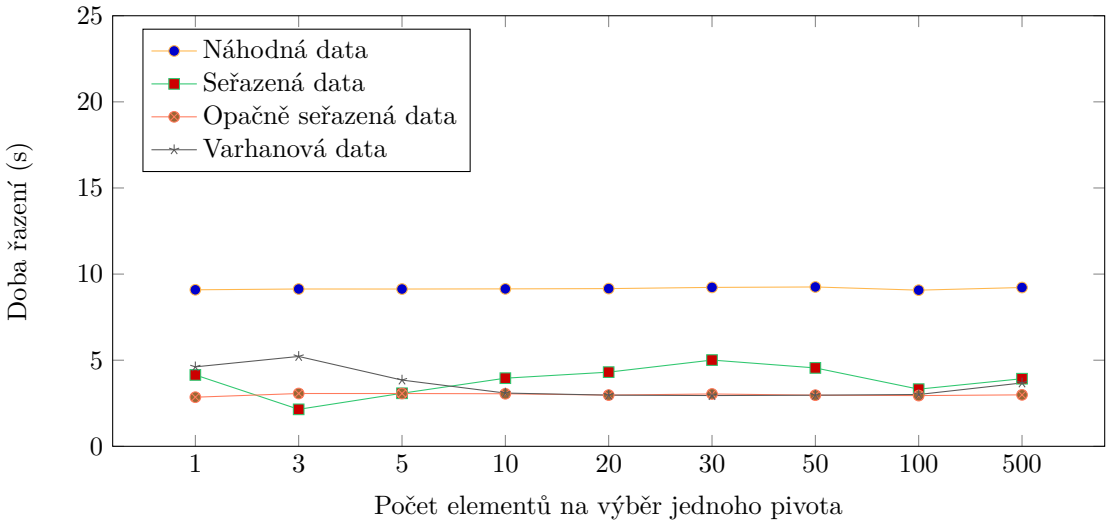
V následující fázi je vstupní pole rozděleno do nejvýše s segmentů a v optimálním případě je $s = c$. Pokud platí, že $s < c$, pak se v této fázi vytvoří práce i pro nevyužitá jádra. Z této skutečnosti plyne, že není potřeba dosahovat vysoké přesnosti výběru pivotů. Nově zavedený parametr nabývá hodnoty 3 a během měření nebyl dále modifikován. Parametr určuje počet elementů na výběr jednoho pivota s tím rozdílem, že v této fázi je počet pivotů fixní (čtyřcestné rozdělování).

V případě čistě sekvenční fáze jsou pivoti vybráni z fixních pozic pole a sampling metodu neuplatňujeme. Délka vstupního pole je již relativně malá a sampling se nadále nevyplatí.

Na grafu 5.4 vidíme malý vliv parametru na dobu řazení algoritmu. Při měření pouze na náhodných datech nebyl vliv na dobu řazení patrný, a proto graf obsahuje i další typy uspořádání dat. Chování algoritmu bylo především u seřazených dat poměrně nepředvídatelné. U takto uspořádaných dat algoritmus dosahoval nejnižší doby běhu pro 3 elementy na výběr jednoho pivota a u vyšších počtů se doba mírně prodlužovala až do hodnoty 30. Naopak všechna ostatní uspořádání dosahovala nejlepších výsledků při parametru hodnoty 100 a pro nižší hodnoty byla jejich doba řazení mírně nestabilní. Z toho důvodu byla hodnota 100 zvolena jako výchozí pro paralelní vícecestné rozdělování.

Mez sekvenčního algoritmu

Dalším parametrem algoritmu je mez přepnutí paralelního řazení na sekvenční. Sekvenční řazení je výhodné pro kratší posloupnosti, kdy by naopak režie vláken mohla mít za následek zpomalení

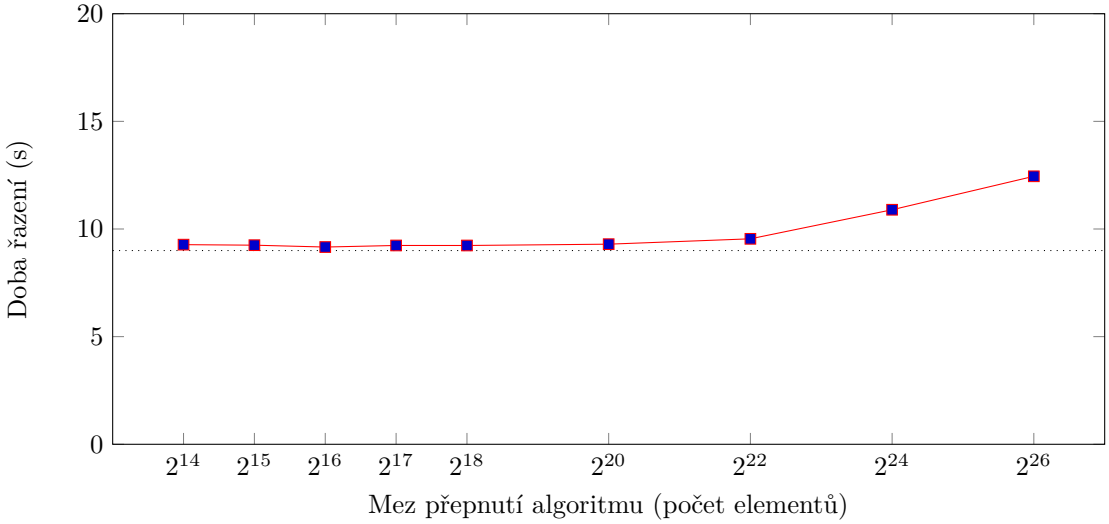


Obrázek 5.4 Vliv počtu elementů na výběr jednoho pivota při řazení dat délky $n = 10^9$ celých čísel (int) a různých typech uspořádání dat.

doby běhu.

K přepnutí dochází během druhé fáze 3.2 paralelního řazení. Měření probíhalo s celkem devíti hodnotami v rozmezí od 2^{14} do 2^{26}. Graf 5.5 zachycuje vyrovnanost rychlosti algoritmu s přibývající velikostí parametru až do 2^{18}. Naopak pro pro větší hodnoty vidíme narůstající zpomalení, které je pro hodnotu 2^{26} až několik sekund.

Z prvních pěti hodnot může být zvolena jakákoli jako výchozí, jelikož jejich vliv na dobu řazení je v podstatě zanedbatelný. My jsme se rozhodli pro mez 2^{18}.

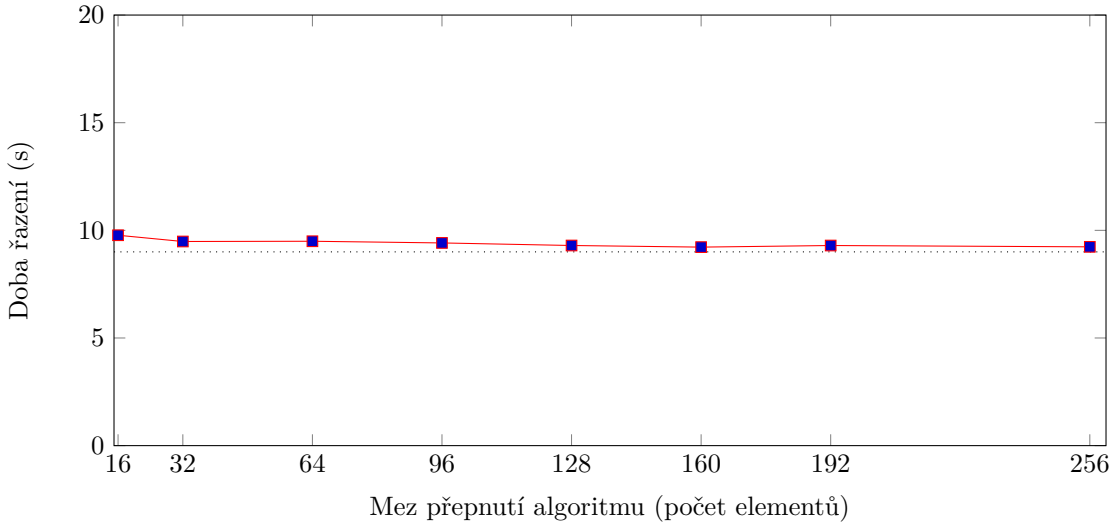


Obrázek 5.5 Vliv na rychlost řazení $n = 10^9$ náhodných celých čísel (int), při různých mezích přepnutí na sekvenční algoritmus.

Mez nerekurzivního algoritmu

Součástí optimalizací sekvenčního algoritmu je mez, kdy se algoritmus přepne na nerekurzivní, pro velmi krátké vstupy. Tím se předejde zbytečnému volání funkce, což má za následek kratší dobu řazení. Také se předejde degradaci na kvadratickou složitost v případě nevhodného výběru pivota, čemuž by se ale mělo předejít již na základě precizního a přesného výběru popsaného v sekci 5.1.

I když se jedná o optimalizaci sekvenční verze, rozhodli jsme se optimální hodnotu parametru zjistit až při kombinaci s paralelním algoritmem. Původní hodnota parametru byla nastavena na 64 elementů, která při měření pouze sekvenční verze Quicksort algoritmu vycházela jako nejvhodnější.



Obrázek 5.6 Vliv velikosti meze pro spuštění Heap-Insertion sort na rychlost algoritmu při řazení $n = 10^9$ náhodných celých čísel (int).

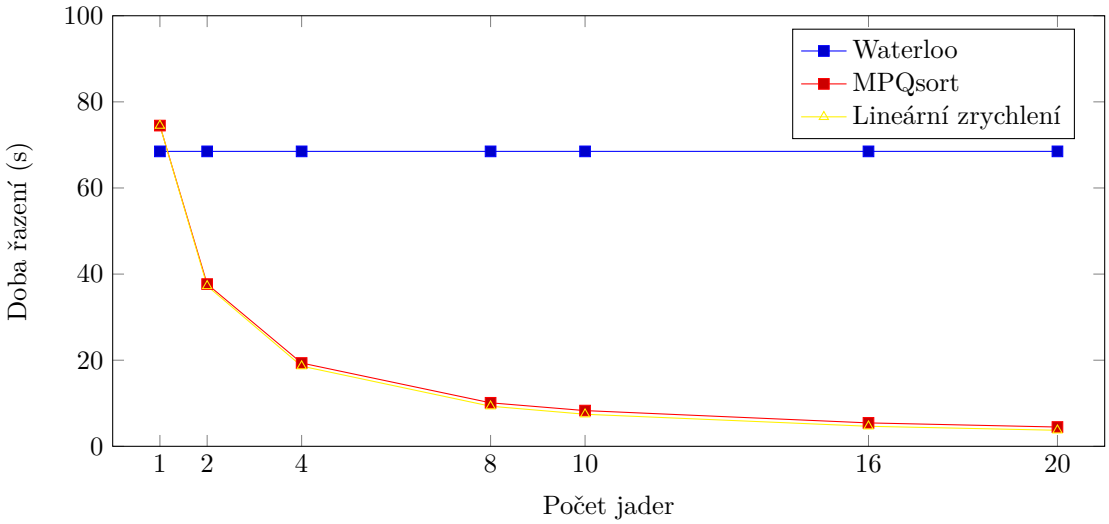
Optimální mez byla volena celkově z osmi různých konfigurací, jejichž vliv na dobu řazení zobrazuje graf 5.6. Vliv tohoto parametru není tak markantní jako u ostatních. Rychlost algoritmu je ve všech případech takřka identická a vidíme pouze mírné navýšení doby běhu pro hodnoty do velikosti 96 včetně. Z toho důvodu volíme 128 elementů jako výchozí hodnotu parametru a bude využita při měření a porovnání s ostatními algoritmy.

Počet využitých jader

Toto měření má za cíl odhalit vliv počtu jader na rychlost řazení algoritmu *MPQsort* a jeho škálovatelnost. Z důvodu porovnání se sekvenční verzí byla volena délka vstupního pole půl miliardy náhodných celých čísel. Tím získáme věrohodné výsledky pro vyhodnocení a zamezíme velmi dlouhé době řazení sekvenčního algoritmu.

Jako sekvenční algoritmus byla zvolena varianta Waterloo, která je i součástí MPQsort algoritmu. Standardní algoritmus STL knihovny *std::sort* neuvažujeme, jelikož obsahuje řadu dalších optimalizací a především není vícecestný.

Výsledky měření pro různý počet jader jsou zobrazeny na grafu 5.7. Pro představu optimálního škálování je v grafu vykreslena křivka lineárního zrychlení, pokud by ho algoritmus dosahoval. Z grafu je patrné, že při využití pouze jednoho jádra je MPQsort pomalejší než sekvenční algoritmus. Konkrétní zpomalení činí 1.09. Naopak již se spuštěním na dvou jádrech vidíme silné zrychlení algoritmu, které je téměř dvojnásobné vůči variantě s jedním jádrem a 1.82



Obrázek 5.7 Škálování algoritmu se zvětšujícím se počet jader při řazení půl miliardy náhodných celých čísel (int). Graf zachycuje i teoretické zrychlení algoritmu a sekvenční Waterloo Quicksort.

krát rychlejší než sekvenční. S narůstajícím počtem jader doba řazení až do deseti jader klesá a téměř ctí lineární zrychlení. Pro deset jader je pouze 1.11 krát pomalejší než lineární zrychlení. Naopak u 16 a 20 jader dochází k mírně výraznějšímu zpomalení a nižšímu škálování. U dvaceti jader MPQsort dosahuje zrychlení 16.6.

Nižší míra škálování pro vyšší počet jader je způsobena velikostí uvažovaného vstupu. Pro kontrolu bylo provedeno měření za cílem porovnat dobu řazení při délce vstupu $n = 2 \cdot 10^9$ a v takovém případě jsme dosáhli až osmnáctinásobného zrychlení.

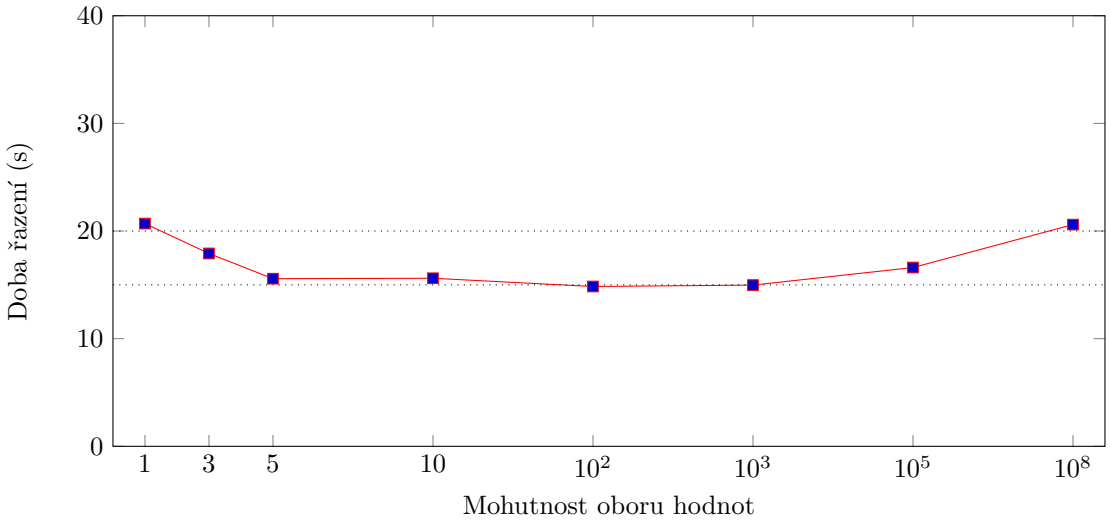
5.2 Otestování efektivity výsledné implementace

V této sekci již pracujeme s výslednou implementací algoritmu včetně nastavených výchozích parametrů pro dosažení nejvyšší efektivity při řazení dat. Cílem je, aby byl algoritmus co nejvíce univerzální a při jakémkoli „nevhodném“ vstupu dosahoval dobrých výsledků. Proto se zde zaměříme na různé konfigurace vstupních dat. Mezi konfigurace patří různé uspořádání elementů ve vstupním poli, různé datové typy a četnost opakujících se prvků (mohutnost oboru hodnot).

Mohutnost oboru hodnot

V rámci měření výkonnosti algoritmu *MPQsort* jsme provedli měření s různými mohutnostmi oboru hodnot. Jednotlivé elementy byly typu int a nabývaly náhodných hodnot v rozsahu 1 až 10^8. Rozsah datového typu int je typicky $2^{32}-1$, ale pro odhalení chování algoritmu nám uvedený rozsah naprosto postačuje.

V grafu 5.8 je zřetelný pokles při mohutnosti oboru hodnot 10^2 vůči mohutnosti 1 v rámci několika sekund. Naopak s nabývající mohutností roste doba řazení algoritmu, což je způsobeno vyšším počtem prohazování elementů mezi segmenty a vyšším počtem porovnání elementů s pivoty. V případě střední mohutnosti je vytvořen menší počet segmentů a z toho důvodu je většina elementů již ve správném segmentu a není potřeba jejich přesunu do jiných. Naopak u velmi malé mohutnosti je vytvářen nedostatečný počet segmentů a dochází k horšímu vyvažování zátěže mezi vlákna.



Obrázek 5.8 Porovnání doby řazení při různých mohutnostech oboru hodnot u $n = 2 \cdot 10^9$ náhodných celých čísel (int).

Z měření vyplývá, že algoritmus pracuje pro malé mohutnosti oboru hodnot optimálně a dosahuje obdobných výsledků jako u větších mohutností. Na základně provedených měření nebyly odhaleny žádné anomálie z pohledu nárůstu doby řazení u malých mohutností. Naopak u středních byl pozorován pokles doby řazení.

Uspořádání vstupních dat

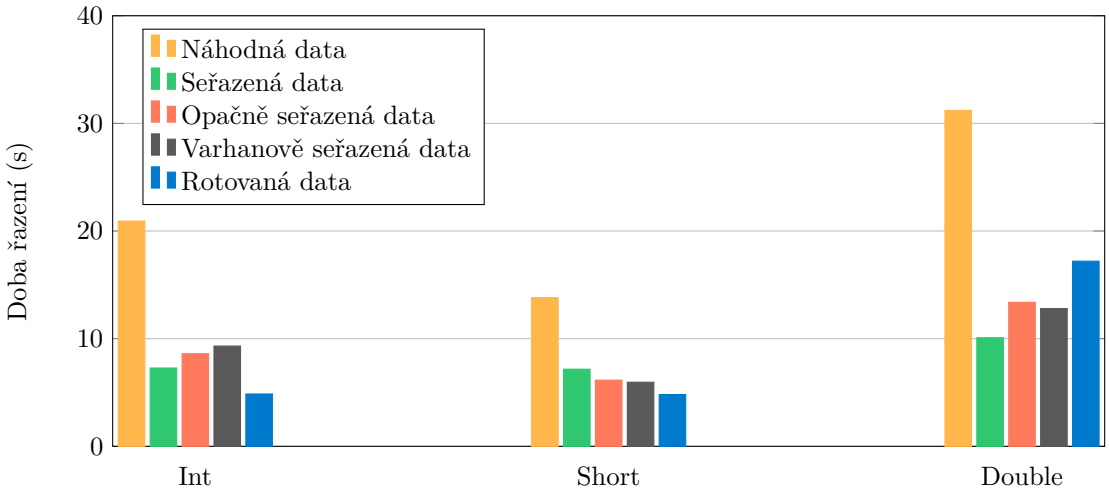
Důležitou součástí vyhodnocení výkonnosti řadícího algoritmu je jeho chování při různých typech uspořádání vstupních dat a datového typu elementů. Pro přehlednost a porovnání vše zobrazíme v jednom grafu.

MPQsort byl testován na číselných datových typech int, short a double. V grafu 5.9 vidíme pokles doby řazení pro datový typ short, který má poloviční velikost v porovnání s typem int. Naopak pro typ double, určený pro desetinná čísla, doba řazení stoupla až o deset sekund. Double bývá naopak větší než datový typ int, a proto je náročnější elementy seřadit. Naměřené hodnoty jsou očekávané a chování algoritmu odpovídá předpokladům.

Uvažované typy seřazení vstupních dat byly probrány v 4.3. Řazení náhodně generovaných dat je zpravidla pro řadící algoritmy nejnáročnější a tato skutečnost je i zachycena v grafu 5.9. Řazení náhodných dat v případě všech datových typů trvá až dvakrát delší dobu než pro ostatní uspořádání. Pro náhodnou posloupnost je využit maximální potenciál algoritmu. Zpravidla musí provést nejvyšší počet operací porovnání a prohození elementů.

V případě seřazené posloupnosti rychlost řazení klesne o více než polovinu u datového typu int a double. Pro short je pokles lehce pod polovinu. U seřazených dat není potřeba elementy prohazovat mezi segmenty a v průběhu algoritmu zůstávají na svém místě. Je však potřeba všechny prvky v poli porovnat s pivoty. Algoritmus by bylo možné případně optimalizovat v případě, že bychom očekávali časté zpracovávání seřazených dat. V přípravné fázi by se zavedla kontrola seřazenosti pole. V takovém případě bychom mohli algoritmus rovnou ukončit a nevstupovat do následujících fází.

Opačně seřazená posloupnost je pro algoritmus mírně náročnější než seřazená. V případě datového typu short bylo naopak řazení rychlejší. Tento typ uspořádání dat je náročný pro algoritmy s nevhodným výběrem pivota, kdy je jako pivot zvolen první a nebo poslední element pole. V takovém případě může u dvoucestných Quicksort algoritmů docházet k nevyváženosti



Obrázek 5.9 Vliv datového typu a uspořádání na dobu řazení $n = 2 \cdot 10^9$ elementů.

segmentů.

Obdobnou rychlost řazení pozorujeme i na varhanově seřazených datech. Vstup může ovlivnit rychlost algoritmu obdobným způsobem jako opačně seřazená data. První polovina pole je korektně seřazena a druhá polovina v přesně opačném pořadí. Opět může ovlivnit výběr pivota, konkrétně způsob výběru jako medián-tří z fixních pozic (první, poslední a prostřední element).

Posledním typem uspořádání jsou rotovaná data. Elementy byly seřazeny a následně cyklicky posunuty o jednu pozici vlevo. Při vhodném výběru pivota byl měla být doba řazení velmi podobná době zpracování seřazené posloupnosti. Naopak při volení pivota jako poslední element může algoritmus silně zpomalit.

Pro všechny typy uspořádání algoritmus *MPQsort* dosahuje skvělých výsledků. Pro žádné uspořádání dat nedošlo ke zpomalení vůči náhodným datům a algoritmus byl naopak v průměru dvakrát rychlejší. Všem výše zmíněným problémům jsme se vyhnuli tím, že výběrem pivota v prvních fázích trávíme delší čas a dáváme si na volbě optimálního pivota záležet. Kromě precizního výběru pivota má nemalý vliv i fakt, že je algoritmus vícecestný a pivotů je veliké množství.

5.3 Porovnání s existujícími implementacemi

V této sekci jsme provedli porovnání výsledné implementace MPQsort s ostatními implementacemi paralelního Quicksortu. Všechny ostatní implementace využívají dvoucestné paralelní rozdělování. Během analýzy dosavadních implementací nebyla nalezena verze implementující vícecestné paralelní rozdělování a z toho důvodu uvádíme pouze verze dvoucestné. Díky této analýze můžeme rozhodnout, zda má nadále smysl zabývat se studií paralelního vícecestného Quicksortu a jestli může vůči dosavadním implementacím přinést zrychlení. V případě sekvenčních verzí se vícecestným rozdělováním algoritmus zrychlil 2.3. Všechny z uvedených algoritmů jsou in-place.

Mezi implementacemi uvažovanými při vyhodnocení jsou:

AQsort
Implementace paralelního Quicksort algoritmu umožňující řazení tzv. *multielementů* [17].

cpp11sort
Implementace založená na C++ vláknech [9].

GNU QS

Implementace ze standardní knihovny libstdc++ od GNU [25].

GNU BQS

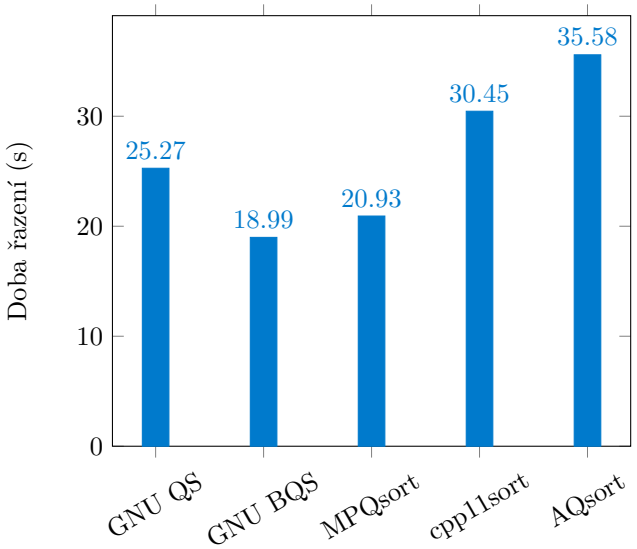
Implementaci ze standardní knihovny libstdc++ od GNU ve vyvážené verzi [25].

AQsort umožňuje řazení multielementů, tedy řazení většího množství polí najednou. Pro řazení se algoritmu poskytne pouze operátor pro prohození a porovnání elementů a není nutná dodatečná znalost o datech či jakékoli jiné operátory. MPQsort si však pri řazení pouze s těmito operátory nevystačí. V průběhu algoritmu je nutné elementy navíc kopírovat a vytvořit pomocné struktury při znalosti datového typu elementu a jeho velikosti. Z toho důvodu nebude vyhodnocení doby řazení multielementů v následujícím textu uvažováno.

V případě implementací od GNU se jedná o široce rozšířené implementace, se kterými se můžeme v praxi setkat. Na algoritmy se budeme v následujícím textu odkazovat přes jejich zkrácené názvy. Porovnání bude probíhat na různých typech uspořádání a datových typech. Zároveň do porovnání zahrneme i měření doby řazení při různých mohutnostech oboru hodnot.

Různé uspořádání a datové typy elementů

Z našeho pohledu je nejdůležitější chování algoritmu na náhodných datech. Níže jsme vytvořili porovnání doby řazení na $n = 2 \cdot 10^9$ náhodných číslech typu int a algoritmy uvedené v předešlé sekci mezi sebou srovnali. Kvůli přehlednosti bude algoritmus MPQsort vždy uprostřed grafu, v levé částí algoritmy GNU a v pravé ostatní.



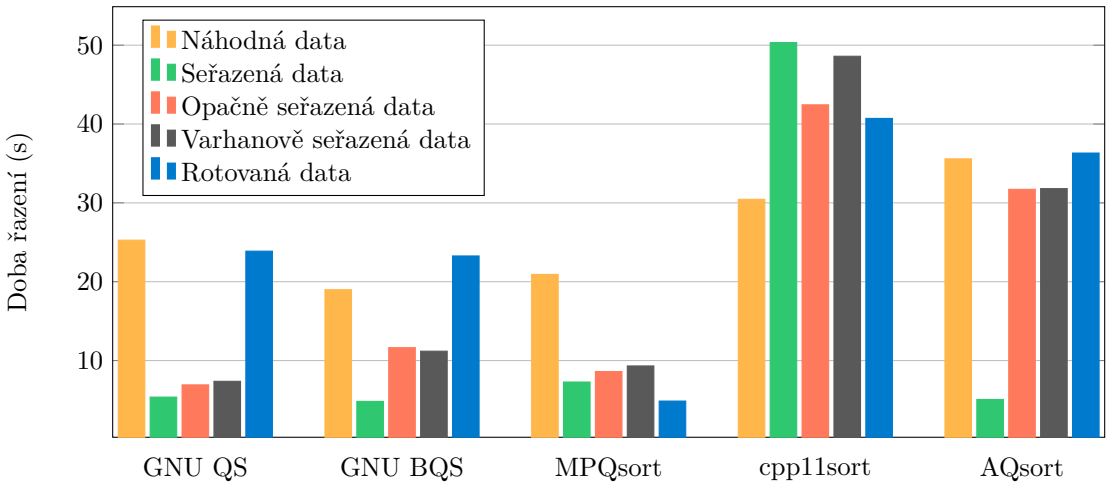
Obrázek 5.10 Porovnání doby řazení $n = 2 \cdot 10^9$ celých čísel (int) s náhodným uspořádáním. Graf zobrazuje porovnání všech uvažovaných algoritmů mezi sebou.

Graf 5.10 ukazuje skvělou efektivitu algoritmu MPQsort vůči ostatním algoritmům. Nejrychlejší z implementací je algoritmus GNU BQS, který je pouze 1.10 krát rychlejší než naše implementace. Celkově se MPQsort umístil na druhém místě v případě řazení náhodných dat a dosahuje zrychlení 1.45 resp. 1.69 vůči pomalejším implementacím cpp11sort a AQsort.

Pro implementaci vícecestného paralelního rozdělování bylo nutné provést přípravnou fázi, alokovat několik pomocných struktur a následně provést i úklidovou fázi.

Po porovnání algoritmů pouze na náhodných datech následuje vyhodnocení jejich rychlosti řazení v případě různých typů uspořádání. Ostatními typy uspořádání jsou seřazená data, opačně

seřazená data, varhanově seřazená data a rotovaná data. Elementy vstupního pole nabývají datových typů int, short a double pro desetinná čísla. Počet elementů v poli je vždy $n = 2 \cdot 10^9$ bez ohledu na paměť, kterou zabírají. Z toho důvodu pravděpodobně budou paměťové nároky elementů typu short menší než v případě int nebo double.



Obrázek 5.11 Porovnání doby řazení $n = 2 \cdot 10^9$ celých čísel (int) s různým uspořádáním. Graf zobrazuje porovnání všech uvažovaných algoritmů mezi sebou.

V případě řazení seřazené posloupnosti elementů typu int dosahovaly nejlepších výsledků algoritmy GNU BQS, AQsort a GNU QS. Jejich doba řazení se liší pouze v rámci stovek milisekund a dosahují velmi dobrých výsledků. O několik sekund horšího výsledku dosahuje MPQsort. Naopak pro cpp11sort seřazená data způsobují velmi výrazné zpomalení několik desítek sekund a doba řazení je 1.65 krát vyšší než v případě náhodných dat. V porovnání s ostatními algoritmy je rychlost MPQsort uspokojující.

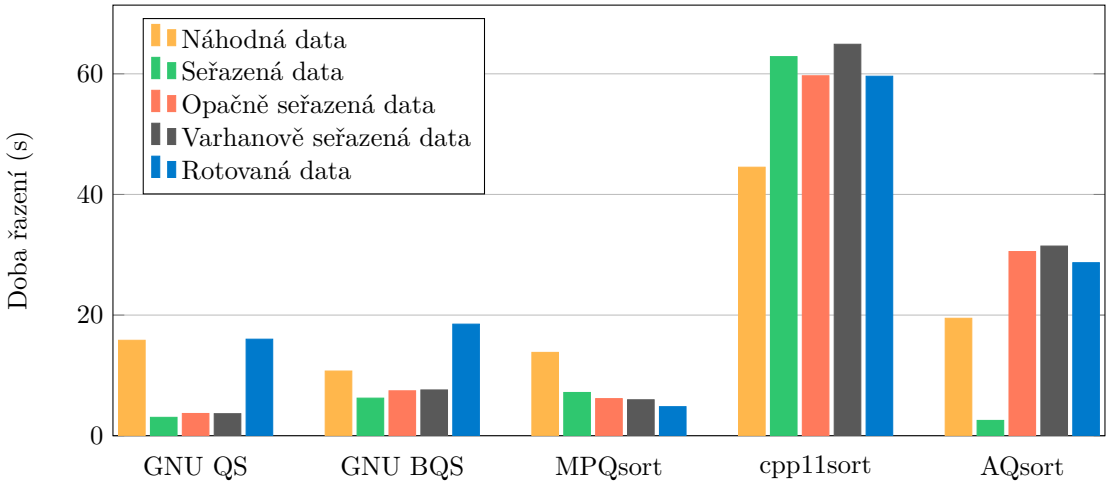
Dalším typem testovacího uspořádání jsou opačně seřazená data. Pro všechny algoritmy, kromě cpp11sort, způsobují zpomalení vůči seřazeným. Nejrychlejším algoritmem je GNU QS, s implementací MPQsort o několik stovek milisekund pomalejší. Implementace tedy dosahuje skvělých výsledků. Naopak AQsort výrazně zpomalil v porovnání se seřazenými daty a dosáhl téměř doby řazení náhodných dat.

Varhanově seřazená data mají na algoritmy velmi podobný vliv jako opačně seřazené pole. Pro skoro všechny algoritmy je doba řazení téměř identická. Výraznější nárůst zaznamenal pouze cpp11sort lišící se o několik sekund.

Posledním typem uspořádání s velmi překvapivými výsledky jsou rotovaná data. Všechny elementy jsou seřazeny a následně cyklicky posunuty o jednu pozici vlevo. MPQsort v tomto případě dosahuje excelentních výsledků a je vůči všem implementacím mnohonásobně rychlejší. Vůči nejrychlejšímu z ostatních algoritmů je naše implementace téměř pětkrát rychlejší. U algoritmů GNU BQS a AQsort se naopak jednalo o nejnáročnější vstup a doba řazení byla větší než u náhodně seřazených dat. Implementace cpp11sort dosahovala lepších výsledků než pro předchozí uspořádání, ale i tak řazení trvalo v porovnání s náhodně seřazenými daty déle.

Na dalších grafech 5.12 a 5.13 jsou ukázány stejné typy uspořádání vstupních dat. V případě datového typu short by algoritmy měly dosahovat kratší doby řazení než u typu int. Zabírá v paměti méně místa a při načítání dat z hlavní paměti jsme schopni do cache pamětí načíst vyšší počet elementů. Naopak typ pro reprezentaci desetinných čísel double zabírá místa více a očekáváme zpomalení.

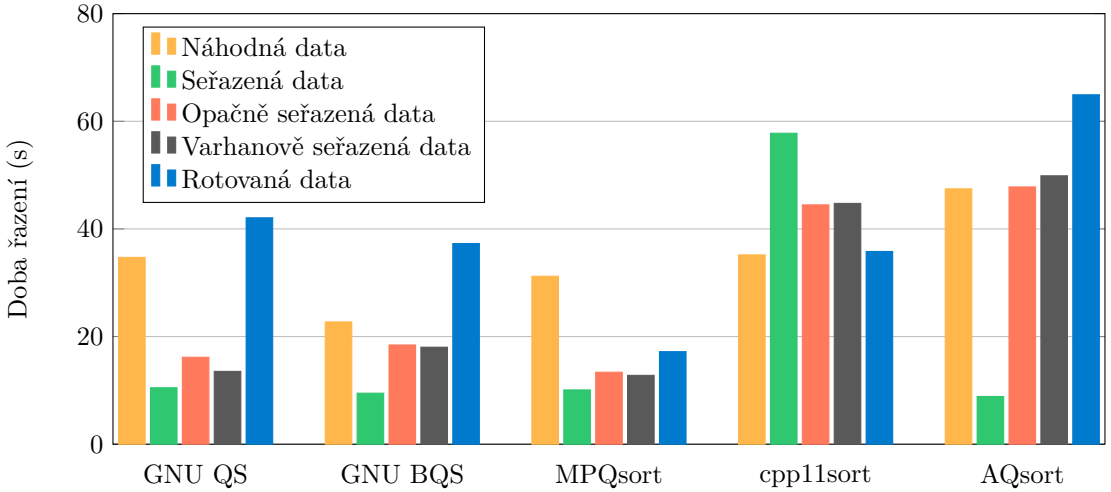
Překvapením u řazení elementů typu short je algoritmus cpp11sort. Všechny ostatní algoritmy dosáhly na menším datovém typu zrychlení, ale tento algoritmus naopak zpomalil ve všech



Obrázek 5.12 Porovnání doby řazení $n = 2 \cdot 10^9$ celých čísel (short) s různým uspořádáním. Graf zobrazuje porovnání všech uvažovaných algoritmů mezi sebou.

uvažovaných typech seřazení vstupních dat.

Vliv uspořádání dat je v zásadě shodný s grafem 5.11. Největší odlišnost pozorujeme u MPQsort, kde je v případě short typu je doba řazení opačně a varhanově seřazených dat kratší v porovnání se seřazenými daty. Rozdíly však nejsou nijak markantní. Naopak u typu double pozorujeme nárůst u rotovaných dat, který je vůči ostatním uspořádáním nejpomalejší. Stále ale platí, že jsou rotovaná data nejrychleji řazena právě naším algoritmem.

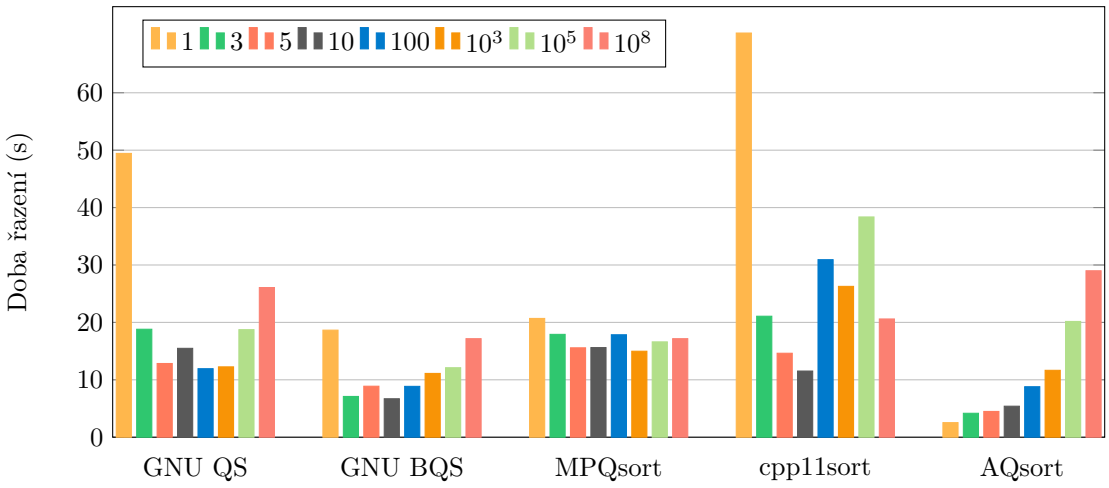


Obrázek 5.13 Porovnání doby řazení $n = 2 \cdot 10^9$ desetinných čísel (double) s různým uspořádáním. Graf zobrazuje porovnání všech uvažovaných algoritmů mezi sebou.

Mohutnost oboru hodnot

Posledním druhem srovnání s ostatními implementacemi je doba řazení pro různé mohutnosti oboru hodnot. O MPQsort víme, že je poměrně stabilní vůči nízkému počtu různých elementů. To ale neznamená, že dosahuje dobrých výsledků a že ostatní implementace pro malé mohutnosti

oboru hodnot nezpracují vstup naopak mnohonásobně rychleji. Proto jsme se rozhodli začlenit i tato maření do kapitoly testování.



Obrázek 5.14 Vliv mohutnosti oboru hodnot na dobu řazení $n = 2 \cdot 10^9$ náhodných celých čísel (int) a porovnání s ostatními implementacemi.

Testování probíhalo na identickém rozsahu mohutností oborů hodnot jako při testování pouze algoritmu MPQsort. V porovnání s ostatními algoritmy se MPQsort jeví jako nejméně senzitivní a poskytuje stabilní dobu řazení dat 5.14. Pro všechny mohutnosti je doba běhu téměř dvacet sekund. Naprosto excelentních výsledků dosahuje AQsort. Vůči MPQsort je v případě mohutnosti 1 osmkrát rychlejší a nejlepším ze všech algoritmů pro malé mohutnosti. Pro mohutnost oboru hodnot nejvýše 10 je ideálním algoritmem a v porovnávaných implementacích nemá konkurenci. Naopak s vyšší mohutností začíná AQsort zpomalovat a pro 10^8 je nejpomalejším.

Při zvolení mohutnosti 1 došlo u algoritmů GNU QS a cpp11sort k velmi výraznému zpomalení. V porovnání s MPQsort byly algoritmy více než dvakrát pomalejší. Naopak s přibývající mohutností doba řazení prudce klesla a naší implementaci se přiblížila.

Závěr

V diplomové práci jsme se zabývali efektivní implementací vícecestného paralelního Quicksort algoritmu pro řazení rozsáhlých polí v jazyce C++. Během rešeršní části práce nebyla nalezena implementace vícecestného paralelního Quicksortu. Všechny dosavadní paralelní verze využívají v paralelním rozdělování pouze jednoho pivota a jsou tedy dvoucestné. Kvůli neexistenci vícecestné paralelní verze nebylo možné navázat na předešlé práce a tím je tato práce unikátní a originální. Vícecestné varianty rozdělování existují pouze u sekvenčních implementací, kde přináší zrychlení doby řazení, což bylo hlavní motivací této práce.

V rešeršní části práce byly probrány různé varianty vícecestných sekvenčních rozdělování pro pochopení základních principů a následně vybrána ta nejvíce efektivní pro využití ve spojení s paralelním algoritmem. Také jsme probrali dosavadní přístupy k paralelnímu dvoucestnému rozdělování. Po studii těchto informací byl navržen nový efektivní in-place algoritmus MPQsort.

Pro paralelizaci MPQsort byla využita rozšířená knihovna OpenMP a její úlohový resp. datový paralelismus. Při návrhu rozhraní byl kladen důraz na jednoduché použití a začlenění do stávajících aplikací. Z toho důvodu má algoritmus rozhraní podobné algoritmům ze standardní knihovny a celá implementace je *header-only*.

Po otestování korektnosti algoritmu MPQsort následovalo rozsáhlé měření rychlosti implementace a následné porovnání s dalšími. Navzdory očekávání dle rešerše implementace MPQsort dosahuje překvapivě velmi dobrých výsledků. Jediný rychlejší algoritmus v případě náhodných dat byl GNU BQS, který byl průměrně o deset procent rychlejší. Ostatní uvedené algoritmy byly pomalejší. Naše implementace zároveň nabízí velmi dobrou stabilitu doby řazení pro různé typy uspořádání vstupních dat a v určitých případech byla i nejrychlejší.

Práce by mohla sloužit jako podklad a být motivací pro výzkum vícecestných paralelních rozdělování v Quicksort algoritmu, jelikož zde demonstrujeme jejich potenciál. Zajímavá by mohla být implementace v prostředí CUDA na grafických kartách, které jsou masivně paralelní.

Bibliografie

1. WILD, Sebastian. *Java 7's Dual Pivot Quicksort* [online]. 2013. Dostupné také z: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-34638. masterthesis. Technische Universität Kaiserslautern.
2. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů* [online]. CZ.NIC, z.s.p.o., 2022. ISBN 978-80-88168-63-8.
3. LEGERSKÝ, Jan. *Základy diskrétní matematiky* [online]. 2022. [cit. 2022-10-20]. Dostupné z: https://courses.fit.cvut.cz/BI-ZDM/lectures. BI-ZDM.
4. KNOP, Dušan; MALÍK, Josef; SUCHÝ, Ondřej; TVRDÍK, Pavel; VALLA, Tomáš. *Algoritmy a grafy 1* [online]. 2022. [cit. 2022-10-20]. Dostupné z: https://courses.fit.cvut.cz/BI-AG1/lectures. BI-AG1.
5. ALEXANDRESCU, Andrei. *Speed Is Found In The Minds Of People* [online]. GitHub repository, 2019 [cit. 2022-10-20]. Dostupné z: https://github.com/CppCon/CppCon2019/blob/master/Presentations/speed_is_found_in_the_minds_of_people/speed_is_found_in_the_minds_of_people__andrei_alexandrescu__cppcon_2019.pdf.
6. WILD, Sebastian. *Dual-pivot quicksort and beyond: Analysis of multiway partitioning and its practical potential* [online]. Technischen Universität Kaiserslautern, 2016. Dostupné také z: https://kluedo.ub.uni-kl.de/frontdoor/deliver/index/docId/4468/file/wild-dissertation.pdf.
7. LANGR, Daniel; ŠIMEČEK, Ivan; TVRDÍK, Pavel. *Paralelní a distribuované programování* [online]. 2022. [cit. 2022-10-20]. Dostupné z: https://courses.fit.cvut.cz/NI-PDP/lectures. NI-PDP.
8. MOHAMMED, Aminu; OTHMAN, Mohamed. Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm. *Information Technology Journal* [online]. 2007, roč. 6. Dostupné z DOI: 10.3923/itj.2007.424.427.
9. SCHOVÁNKOVÁ, Klára. *Paralelní řazení v C++11* [online]. 2019. Dostupné také z: https://dspace.cvut.cz/bitstream/handle/10467/82317/F8-DP-2019-Schovankova-Klara-thesis.pdf. Dipl. pr. ČVUT Fakulta informačních technologií v Praze.
10. LATIF, Abdel; ABU DALHOUM, Abdel; KOBBAEY, Thaeer; SLEIT, Azzam; ALFONSECA, Manuel; DE LA PUENTE, Alfonso. Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique. *Wulfenia* [online]. 2012, roč. 19, s. 543–552 [cit. 2022-10-18]. Dostupné z: https://www.researchgate.net/publication/235351491_Enhancing_QuickSort_Algorithm_using_a_Dynamic_Pivot_Selection_Technique.
11. SEDGEWICK, Robert. *Quicksort*. 1980. ISBN 0-8240-4417-7. Dipl. pr. Stanford University. Dotisk autorovo práce.

12. KUSHAGRA, Shrinu; LOPEZ-ORTIZ, Alejandro; MUNRO, J. Ian; QIAO, Aurick. *Milti-Pivot Quicksort: Theory and Experiments* [online]. 2013. [cit. 2022-11-19]. Dostupné z: https://epubs.siam.org/doi/abs/10.1137/1.9781611973198.6.

13. TERPSTRA, Dan; JAGODE, Heike; YOU, Haihang; DONGARRA, Jack. Collecting Performance Data with PAPI-C. In: MÜLLER, Matthias S.; RESCH, Michael M.; SCHULZ, Alexander; NAGEL, Wolfgang E. (ed.). *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11261-4.

14. BOARD, OpenMP Architecture Review. *OpenMP API specification for parallel programming version 4.5* [online]. 2022. [cit. 2022-10-13]. Dostupné z: https://www.openmp.org/.

15. TSIGAS, P.; ZHANG, Y. *A Simple, Fast Parallel Implementation of Quicksort And Its Performance Evaluation on SUN Enterprise 10000.* 2003. ISBN 0-7695-1875-3.

16. FRANCIS, R.; PANNAN, L. *A parallel partition for enhanced parallel Quicksort* [online]. Parallel Computing, 1992 [cit. 2022-10-26].

17. LANGR, Daniel; TVRDIK, Pavel; SIMECEK, Ivan. AQsort: Scalable Multi-Array In-Place Sorting with OpenMP. *Scalable Computing: Practice and Experience* [online]. 2016, roč. 17. Dostupné z DOI: 10.12694/scpe.v17i4.1207.

18. GOOGLE; COMMUNITY. *Google/benchmark: GoogleTest - Google Testing and mocking framework* [online]. GitHub, 2022 [cit. 2022-08-21]. Dostupné z: https://github.com/google/googletest.

19. COMMUNITY. *Clang 16.0.0git documentation* [online]. 2022. [cit. 2022-09-08]. Dostupné z: https://clang.llvm.org/docs/index.html.

20. GOOGLE; COMMUNITY. *Google/googletest: GoogleBenchmark - A microbenchmark support library* [online]. GitHub, 2022 [cit. 2022-10-02]. Dostupné z: https://github.com/google/benchmark.

21. ROZENTAL, Gennadiy; ENFICIAUD, Raffi. *Boost.Test* [online]. 2022. [cit. 2022-08-21]. Dostupné z: https://www.boost.org/doc/libs/1_68_0/libs/test/doc/html/index.html.

22. COMMUNITY. *Doctest/Doctest: The fastest feature-rich C++11/14/17/20 single-header testing framework* [online]. GitHub, 2022 [cit. 2022-08-21]. Dostupné z: https://github.com/doctest/doctest.

23. CATCHORG; COMMUNITY. *Catchorg/Catch2: A modern, c++-native, Test Framework for unit-tests, TDD and BDD - using C++14, C++17 and later* [online]. GitHub, 2022 [cit. 2022-08-21]. Dostupné z: https://github.com/catchorg/Catch2.

24. *C and C++ reference* [online]. 2022. [cit. 2022-09-02]. Dostupné z: https://en.cppreference.com/w/.

25. *GNU Standard C++ Library* [online]. Free Software Foundation, Inc., 2022. [cit. 2022]. Dostupné z: https://gcc.gnu.org/onlinedocs/libstdc++/.

Obsah přiloženého média