FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Paralelní řazení v C++11
Student:	Bc. Klára Schovánková
Vedoucí:	Ing. Daniel Langr, Ph.D.
Studijní program:	Informatika
Studijní obor:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s problematikou paralelního řazení dat v prostředí více-vláknových aplikací se sdílenou pamětí. Prostudujte paralelní řadící algoritmy a jejich více-vláknové implementace, především ty určené pro programovací jazyk C++. Zaměřte se na tzv. "in-place" řazení, konkrétně na paralelní verze algoritmu quicksort a možnosti jeho kombinace s dalšími algoritmy (insertion sort, heapsort, atd.). Navrhněte paralelní verzi algoritmu quicksort, která bude implementovatelná pouze na základě C++11 vláken a příslušných synchronizačních technik (atomické proměnné, mutexy, apod.), tj. bez využití nadstandardních rozšíření jazyka C++ (jako je např OpenMP) a externích knihoven (jako je např. Intel TBB). Vytvořte efektivní implementaci navrženého algoritmu a porovnejte ji experimentálně s existujícími implementacemi paralelních in-place algoritmů dostupných pro C++.

Seznam odborné literatury

Dodá vedoucí práce.

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

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 15. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Paralelní řazení v C++11

Bc. Klára Schovánková

Katedra teoretické informatiky
Vedoucí práce: Ing. Daniel Langr, Ph.D.

7. května 2019

Poděkování

Děkuji vedoucímu práce Ing. Danielu Langrovi, Ph.D. za cenné rady a čas, který mi věnoval při psaní této práce.

Dále děkuji své rodině za podporu v průběhu celého studia a zejména během vytváření této práce.

Prohlášení

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

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

V Praze dne 7. května 2019 .....

Odkaz na tuto práci

Schovánková, Klára. *Paralelní řazení v C++11*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

V této práci je představen C++11Sort, nová paralelní verze in-place algoritmu quicksort. C++11Sort je implementován pouze na základě C++11 vláken a příslušných synchronizačních technik, tedy bez použití nadstandardních rozšíření jazyka C++ a externích knihoven. V rešeršní části jsou popsány existující implementace paralelní verze algoritmu quicksort. Tyto existující implementace jsou následně experimentálně porovnány s vytvořeným algoritmem C++11Sort. C++11Sort dosahuje v experimentech výborných výsledků, například při řazení dvou miliard celých čísel je o 28 % rychlejší než nejrychlejší existující implementace.

Klíčová slova paralelní řazení, C++11, quicksort, fond vláken, in-place řazení, paralelní rozdělování

Abstract

A new parallel variant of the quicksort algorithm called C++11Sort is presented in this thesis. C++11Sort is implemented based on the C++11 threads and the respective synchronization techniques, i.e. without the use of the non-standard C++ extensions and external libraries. In the search section, existing implementations of the parallel versions of the quicksort algorithm are described. These implementations are afterwards experimentally compared to the C++11Sort algorithm. C++11Sort achieves excellent results in those experiments. When sorting two billions of integers, it is faster than the fastest existing implementation by 28 %.

Keywords parallel sorting, C++11, quicksort, thread pool, in-place sorting, parallel partitioning

Obsah

Seznam obrázků

Úvod

Řazení je nedílnou součástí mnoha používaných algoritmů. Protože se často podílí významnou mírou na době běhu algoritmu, je vhodné, aby jeho implementace byla co nejefektivnější. V současné době jsou téměř všechny osobní počítače více-jádrové. Aby mohla být využita výpočetní síla hardwaru, je užitečné řadící algoritmy paralelizovat.

V programovacím jazyce C++ je paralelizace algoritmů, tedy i paralelizace řadícího algoritmu, součástí standardu od verze 17. Bohužel v současné době není tento standard implementován ve významných překladačích, například v překladačích GCC a Clang. Existence paralelních řadících algoritmů v implementacích standardních knihoven těchto překladačů by vývojářům umožnila využívat lépe výpočetní sílu hardwaru bez nutnosti hlubší znalosti paralelizace.

Paralelizace řadících algoritmů je v současné době podporována pouze překladači MSVC a Intel C++, které však používají rozsáhlé knihovny specifické pouze pro tyto překladače. Pokud chtějí vývojáři použít paralelní řadící algoritmy a překládat své programy jinými překladači, musejí použít některá nadstandardní rozšíření jazyka C++ nebo rozsáhle externí knihovny (například Intel TBB).

Tato práce je zaměřena na tzv. „in-place“ řazení v prostředí více-vláknových aplikací se sdílenou pamětí. Konkrétně je tato práce zaměřena na paralelní verze algoritmu quicksort. Existující implementace paralelních verzí algoritmu quicksort nejčastěji používají OpenMP, což je nadstandardní rozšíření jazyka C++. Během rešeršní práce nebyla nalezena implementace paralelní verze algoritmu quicksort, která by nevyužívala nadstandardní rozšíření nebo externí knihovny a zároveň by byla efektivní.

Cílem této práce je návrh efektivní paralelní verze algoritmu quicksort, která je implementovatelná pouze na základě C++11 vláken a příslušných synchronizačních technik, tj. bez použití nadstandardních rozšíření jazyka C++ a externích knihoven. Dalším krokem je implementace navrženého algoritmu

a jeho experimentální porovnání s existujícími implementacemi paralelních in-place verzí algoritmu quicksort dostupných pro C++.

Kapitola 1

Řadící algoritmy

Řadící algoritmy slouží k uspořádání dat do určitého pořadí. Se seřazenými daty se v mnohých případech lépe pracuje, například je v nich možné vyhledávat rychleji než v datech neseřazených. Řazení dat může v některých případech zabírat podstatnou část doby běhu programu, proto je jeho optimalizace velmi důležitá.

Problém řazení dat je možné formálněji popsat následovně. Je dána posloupnost dat $A = (a_1, a_2, \ldots, a_n)$. Výstupem řadícího algoritmu je permutace vstupních dat $A' = (a'_1, a'_2, \ldots, a'_n)$, která je neklesající. Tedy pro každý prvek platí, že je větší než předchozí prvek (nebo je mu roven), podle požadovaného úplného uspořádání: $a'_1 \leq a'_2 \ldots \leq a'_n$.

Na zařízeních, která podporují více-vláknové zpracování, je možné dosáhnout podstatného zrychlení řazení paralelizací řadících algoritmů.

V této kapitole jsou popsány základní vlastnosti řadících algoritmů. Dále je zde představen algoritmus quicksort, na jehož efektivní paralelní implementaci je tato práce zaměřena. Je zde vysvětlena základní myšlenka sekvenčního algoritmu quicksort společně s možnostmi jeho efektivní implementace. V závěru kapitoly je popsán princip paralelní verze algoritmu quicksort.

1.1 Vlastnosti řadících algoritmů

Řadících algoritmů existuje velké množství. Je možné je rozlišovat a kategorizovat na základě jejich vlastností. Mezi nejvýznamnější vlastnosti řadících algoritmů patří jejich časová a paměťová složitost, stabilita a datová citlivost [1].

1.1.1 Časová složitost

Časová složitost vyjadřuje závislost délky běhu algoritmu na počtu řazených dat. Obvykle je používána asymptotická časová složitost, u které jsou za-

nedbány multiplikativní a aditivní konstanty. Při zjišťování časové složitosti řadících algoritmů je zajímavé sledovat složitost v průměrném a v nejhorším případě.

Běžně používané sekvenční řadící algoritmy pro všeobecné užití, mezi které patří například merge sort, heapsort a quicksort v průměrném případě, mají časovou složitost $\mathcal{O}(n \cdot \log(n))$. Algoritmy založené na binárním porovnávání hodnot použijí alespoň $\Theta(n \cdot \log(n))$ porovnání [1]. Proto není možné, aby měl algoritmus založený na binárním porovnávání hodnot složitost menší než právě $\mathcal{O}(n \cdot \log(n))$.

Algoritmy fungující na jiném principu než na binárním porovnávání dvou hodnot, mohou mít nižší složitost. Například counting sort a radix sort mají složitost lineární. Tyto algoritmy však vyžadují, aby řazená data měla nějakou speciální vlastnost, proto nejsou vhodné pro všeobecné použití. Speciální vlastností může být například omezený počet nabývaných hodnot.

1.1.2 Paměťová složitost

Paměťová složitost vyjadřuje závislost potřebné dodatečné paměti k provedení výpočtu na množství vstupních dat. Do paměťové složitosti algoritmu se nezapočítávají paměťové buňky, v nichž jsou zadána vstupní data. Zajímavá je hlavně paměťová složitost operací nad již načtenými daty. V závislosti na množství použité dodatečné paměti jsou rozlišovány dva typy algoritmů, a to *in-place* (na místě) algoritmy a *out-of-place* (mimo místo) algoritmy.

Řadící algoritmy typu *in-place* zpracovávají vstup prakticky bez použití dodatečné paměti. Definice in-place algoritmu se mezi jednotlivými autory mírně liší. Někdy je požadováno, aby algoritmus ke svému běhu potřeboval pouze konstantní množství paměti, které nezávisí na velikosti vstupu. Jindy je považován za in-place takový algoritmus, který potřebuje $o(n)$ dodatečné paměti. Mezi in-place algoritmy patří například bubble sort, který k práci potřebuje pouze konstantní množství paměti.

Řadící algoritmy typu *out-of-place* potřebují k řazení větší množství pomocné paměti než algoritmy typu in-place. Velikost potřebné dodatečné paměti je závislá na velikosti vstupních dat. Typickým příkladem out-of-place algoritmu je merge sort, který potřebuje navíc $\mathcal{O}(n)$ paměti, protože pracuje s pomocným polem, které je stejně velké jako vstupní pole.

1.1.3 Stabilita řazení

Řazení se nazývá stabilní, pokud v jeho průběhu nedojde k prohození prvků se stejnou hodnotou. Tedy pokud pro dva prvky řazené posloupnosti platí, že $a_i = a_j$ pro indexy $i < j$, pak se prvek a_i vyskytuje ve výstupní posloupnosti před prvkem a_j.

Stabilita řadícího algoritmu může být důležitá, pokud je například řazena posloupnost nejprve podle jednoho klíče a následně podle druhého klíče, jako

je tomu například při řazení jmen. Algoritmy bubble sort a insert sort jsou příklady stabilních algoritmů, zatímco algoritmus selection sort je nestabilní.

1.1.4 Datová citlivost

Řadící algoritmus je považován za datově citlivý, pokud počet operací potřebných k seřazení závisí nejen na množství dat, ale i na jejich vstupním uspořádání. Datově citlivým algoritmem může být například bubble sort, pokud při každém průchodu kontroluje, jestli byl alespoň jeden prvek prohozen. Oproti tomu například merge sort je datově necitlivý, protože seřadí každou permutaci dat stejně rychle.

1.2 Quicksort

V současné době jsou implementace in-place řadících algoritmů pro všeobecné použití obvykle založeny na algoritmu quicksort. I u většiny překladačů se v roli standardní funkce pro řazení většinou vyskytuje právě quicksort. Jeho správná implementace v praxi běží (v průměrném případě a na velkých datech) rychleji než implementace jiných řadících algoritmů [1]. I proto je tato práce věnována efektivní paralelní implementaci právě algoritmu quicksort.

1.2.1 Popis algoritmu

Quicksort je řadící algoritmus založený na principu porovnej a prohoď. Jeho základem je rozdělení řazené posloupnosti na dvě části. Při rozdělování jsou prvky posloupnosti přeuspořádány tak, že v první části jsou prvky menší než určitá zvolená hodnota (označována názvem „pivot“) a ve druhé části jsou prvky větší (nebo rovny). V ideálním případě je pivot zvolen tak, že jsou obě části stejně velké. Následně je rekurzivně zavolán quicksort na obě vzniklé části. Jestliže jsou rekurzivně seřazeny obě části, vznikne seřazená posloupnost. Pseudokód naivního quicksortu a rozdělení je znázorněn v algoritmu 1.

Přestože je časová složitost algoritmu quicksort v nejhorším případě $\mathcal{O}(n^2)$, v průměrném případě dosahuje složitosti $\mathcal{O}(n \cdot \log(n))$. V praxi je navíc možné se nejhoršímu případu vyhnout, jak je popsáno v sekci 1.2.5.

Z hlediska paměťové složitosti se quicksort obvykle řadí mezi in-place algoritmy. Quicksort sice potřebuje místo na zásobníku pro rekurzivní volání, ale potřebná paměť je asymptoticky menší než paměť potřebná pro běh řadících algoritmů typu out-of-place, mezi které patří například merge sort. V případě správné implementace s omezením hloubky rekurze je potřebná paměť pro volání, jak je popsáno v sekci 1.2.3, pouze logaritmická vzhledem k délce řazené posloupnosti.

Quicksort není v základní verzi stabilní, protože během rozdělování jsou prvky prohazovány tak, že jejich vstupní pořadí nemusí být zachováno.

Algoritmus 1 Quicksort

```
1: function QUICKSORT(A, lo, hi)
2:     if lo < hi then
3:         p ← partition(A, lo, hi)
4:         quicksort(A, lo, p − 1)
5:         quicksort(A, p + 1, hi)
6: function PARTITION(A, lo, hi)
7:     pivot ← A[hi]
8:     i ← lo
9:     for j ← lo to hi − 1 do
10:        if A[j] < pivot then
11:            swap A[i] with A[j]
12:            i ← i + 1
13:    swap A[i] with A[hi]
14:    return i
```

Algoritmus quicksort je datově citlivý. Uspořádání dat má především vliv na výběr pivota, tedy v důsledku na rovnoměrné rozdělení posloupnosti na dvě části.

1.2.2 Výběr pivota

Správný výběr pivota je klíčovým prvkem efektivní implementace quicksortu. Jestliže je pivotem zvolen prvek blízký mediánu, je algoritmus velmi rychlý, protože je posloupnost rozdělena rovnoměrně na dvě části. V opačném případě, pokud je například pivotem vždy maximální prvek, může být časová složitost algoritmu až $\mathcal{O}(n^2)$.

Zejména v paralelní verzi je správný výběr pivota důležitý pro vhodné vyvážení zátěže jednotlivých jader procesoru.

V praxi je možné se setkat s různými postupy, jak určit, který prvek posloupnosti by měl být pivotem [2]. Naivním řešením je výběr prvku na některé fixní pozici. Je možné například vybrat vždy poslední prvek posloupnosti. Tato metoda je však hodně datově citlivá a může být velmi pomalá pro již částečně seřazená data. Pokud je použita tato metoda výběru pivota, může být docíleno zachování efektivity algoritmu i v případě speciálního rozložení dat (např. již seřazená posloupnost) náhodným přeuspořádáním řazené posloupnosti před spuštěním samotného řadícího algoritmu.

Druhou možností je určit jako pivota náhodný prvek. Tato metoda je méně citlivá na částečně seřazená data, ale stále je zde dost vysoká pravděpodobnost špatné volby pivota.

Další možným postupem je výběr K prvků posloupnosti, jejich seřazení a zjištění jejich mediánu. Tento medián je následně použit jako pivot pro celou řazenou posloupnost. Počet vybraných prvků K je volen s ohledem na

velikost a povahu řazených dat. Medián vyššího počtu prvků je pravděpodobně bližší mediánu celé posloupnosti. Na druhou stranu je nutné počítat s časem nutným pro seřazení vybraných prvků při hledání jejich mediánu, který roste s vyšším K.

V praxi se často používá medián tří prvků nebo medián tří mediánů ze tří prvků. Podle článku Mohammeda Amina [2] je nejlepších výsledků dosaženo při použití mediánu pěti vybraných prvků.

1.2.3 Omezení hloubky rekurze

Jedním z možných vylepšení algoritmu je omezení hloubky rekurze. Každé rekurzivní volání quicksortu spotřebovává paměť zásobníku. Počet rekurzivních volání závisí na volbě pivota. Pokud je pivot volen nevhodně (např. je jím vždy maximální prvek), může dojít až k lineárnímu počtu rekurzivních volání. Nutnost použití lineárního množství dodatečné paměti může vést až k pádu programu.

Proto je vhodné omezit hloubku rekurzivního volání. Optimalizace spočívá v tom, že menší část rozděleného pole je seřazena rekurzivně. Větší část je poté seřazena iterativně. Tím je zaručeno, že hloubka rekurze (a tedy i použitá dodatečná paměť) bude vždy nejvýše $\mathcal{O}(log_2(n))$.

1.2.4 Kombinace s algoritmem insert sort

Přístup rozděl a panuj je vhodný pro velké posloupnosti. Pro malé posloupnosti, čítající pouze několik prvků, již tolik optimální není. Proto je někdy quicksort upraven tak, že pokud počet prvků v řazené sekvenci klesne pod určitou mez, je aplikován jiný, obvykle nerekurzivní, řadící algoritmus vhodnější pro menší sekvence. Jako tento koncový algoritmus je často volen insert sort.

Insert sort je stabilní, datově citlivý řadící algoritmus, který nevyžaduje pro řazení dodatečnou paměť. Jeho složitost je v průměrném případě téměř kvadratická. Proto se nehodí k řazení velkých dat. Naopak vyniká v případech, kdy je posloupnosti již částečně seřazena nebo pokud je posloupnost malá a obsahuje maximálně desítky prvků. Proto se právě insert sort používá jako pomocná řadící funkce u složitějších algoritmů.

Insert sort funguje na principu vkládání. Řazená posloupnost se dělí na seřezanou a neseřazenou část. Bez újmy na obecnosti je možné předpokládat, že seřazená část je v levé části posloupnosti. Vkládání probíhá tak, že je nejlevější prvek z neseřazené části porovnáván s prvkem, který se nachází na pozici před ním. Dokud je vkládaný prvek menší než prvek ze seřazené části, jejich hodnoty se prohodí a vkládaný prvek je porovnáván s dalším prvkem seřazené části. Když je vkládaný prvek na správné pozici, velikost neseřazené části je zmenšena o jedna a je následně vkládán další prvek, dokud není neseřazená část prázdná.

Jinou variantou výše popsané kombinace je ta, při které quicksort nepokračuje v řazení, pokud je velikost řazené posloupnosti menší než určitá mez k. Poté, co je takto zpracováno celé pole, je pole k-seřazeno. To znamená, že každý prvek je nejvýše k pozic od své cílové pozice v seřazené posloupnosti. Pokud je následně spuštěn nad k-seřazeným polem insert sort, je jeho složitost pouze $\mathcal{O}(kn)$ [3]. Doba nutná ke koncovému seřazení je tedy pouze lineární, za předpokladu, že k je konstanta. V porovnání s předchozí variantou tato verze provede méně instrukcí, protože insert sort je zavolán pouze jednou. Na moderních počítačích však může být, podle zmíněného článku, tento přístup spíše kontraproduktivní, vzhledem k neoptimálnímu použití mezipamětí.

1.2.5 Kombinace s algoritmem heapsort

Nevýhodou quicksortu je, že může mít v nejhorším případě až kvadratickou časovou složitost. V praxi je důležité se vyvarovat toho, aby algoritmus běžel příliš dlouho. Proto se často quicksort kombinuje s jiným řadícím algoritmem, který je také vhodný pro velká data a má garantovanou $\mathcal{O}(n \cdot log(n))$ složitost.

Tímto algoritmem bývá často heapsort. Heapsort je in-place nestabilní řadící algoritmus založený na principu porovnej a prohoď. Heapsort je v praxi obvykle pomalejší než quicksort [1]. Jeho výhodou však je, že jeho složitost je i v nejhorším případě $\mathcal{O}(n \cdot log(n))$.

Pokud quicksort dosáhne přílišného zanoření, přepne se na heapsort. Hranice pro přepnutí je obvykle založena na logaritmu počtu řazených prvků. Včasným přepnutím na řadící algoritmus s garantovanou $\mathcal{O}(n \cdot log(n))$ složitostí je možné zabránit tomu, aby quicksort běžel v kvadratickém čase. Této kombinaci se někdy říká introsort. V praxi dosahuje kombinace těchto dvou algoritmů v průměrném případě podobné výkonnosti jako quicksort.

1.2.6 Rozdělení na tři části

V další variantě quicksortu nedochází k dělení vstupní posloupnosti na dvě části, ale na tři. V první části jsou prvky menší než pivot, ve druhé prvky, které jsou rovny pivotovi a ve třetí části jsou prvky větší než pivot. Toto dělení je zvláště výhodné u posloupností, ve kterých je počet nabývaných hodnot malý. Část s prvky, které jsou rovny pivotovi totiž nemusí být dále vůbec řazena, čímž je urychlen celý běh programu. Pro posloupnosti s velkým počtem nabývaných hodnot není rozdělení na tři části přínosné, protože posloupnost neobsahuje větší množství prvků, které by byly pivotovi rovny.

1.3 Paralelní quicksort

Podstatou quicksortu je rozdělení řazené posloupnosti na dvě menší podposloupnosti a rekurzivní zavolání řazení těchto menších posloupností. Rekurzivní volání jsou ve více-vláknovém prostředí přirozenými kandidáty na

úlohový paralelismus. Podposloupnosti mohou být zpracovávány samostatně, jejich řazení je na sobě nezávislé. Quicksort se tedy jeví být dobrým kandidátem na paralelizaci.

Kromě samotného paralelního řazení menších částí posloupnosti je možné také paralelně přistupovat k rozdělení posloupnosti na prvky menší a větší než je pivot.

1.3.1 Paralelní řazení podposloupností

Paralelní řazení již rozdělených menších podposloupností je relativně přímočaré. Nechť p označuje počet vláken, která vstupní posloupnost paralelně zpracovávají. V optimálním případě je řazená posloupnost rozdělena rovnoměrně na p částí a každé vlákno zpracovává samostatně svoji část.

Problém nastává v tom, jak posloupnost mezi jednotlivá vlákna rozdělit, aby všechna jádra procesoru dostala přiřazený stejný objem práce. Při rozdělování posloupnosti pivot obvykle není přímo medián, takže jednotlivé vzniklé podposloupnosti mohou být různě velké. Navíc potřebný výpočetní čas k seřazení i stejně dlouhých podposloupností může být různý. Z těchto důvodů je v efektivní implementaci nutné provádět vyvažování zátěže.

Vyvažování zátěže může být realizováno například rozdělením vstupní posloupnosti na větší počet podposloupností, tedy větší počet úloh než je počet vláken. Jednotlivé úlohy mohou být následně vláknům přiřazovány dynamicky, když dokončí provádění úlohy předchozí.

Alternativně je možné vyvažovat zátěž tak, že vlákna mohou po dokončení vlastní práce „krást“ práci jiným vláknům. Při tomto přístupu si jednotlivá vlákna ukládají své dosud nezpracované podposloupnosti na zásobník. Pokud nějaké vlákno dokončí svoji práci, převezme kus práce ze zásobníku jiného vlákna.

1.3.2 Paralelizace rozdělování posloupnosti

Aby bylo možné provádět paralelní řazení menších podposloupností, je nutné nejprve několikrát rekurzivně rozdělit posloupnost na prvky menší a větší než pivot. Časová složitost sekvenčního rozdělování je lineární vzhledem k délce posloupnosti.

Sekvenční rozdělování na nejvyšších úrovních rekurze by výrazně zamezovalo škálovatelnosti paralelního quicksortu [4]. Z tohoto důvodu je nutné provádět rozdělování posloupnosti paralelně, alespoň na nejvyšších úrovních rekurze. Paralelizace této části quicksortu je náročnější a je podrobně popsána v kapitole 3. Při rozdělování je třeba vzít v úvahu například i to, jak vlákna přistupují do paměti, aby nedocházelo k tzv. falešnému sdílení.

Kapitola 2

Implementace paralelního quicksortu v jazyce C++

V současné době jsou téměř všechny osobní počítače více-jádrové. Výpočetní síla hardwaru je však často nevyužita, kvůli nedostatečně paralelizovanému softwaru.

Paralelizace programů je obecně obtížná. Většina nástrojů pro vytváření paralelních programů je primitivní, v porovnání s existujícími nástroji pro tradiční sekvenční vývoj [5]. Psaní paralelních algoritmů je také náchylné k chybám. Na rozdíl od sekvenčního vývoje je nutné myslet na synchronizaci jednotlivých výpočtů a na to, jak přistupují do paměti a žádají o prostředky. Špatné použití synchronizačních technik může vést například k uváznutí či k nedefinovanému chování.

Proto je vhodné, aby programovací jazyk poskytoval programátorům jednoduchý způsob, jak paralelizovat výpočetně náročné části programů. Pro práci s velkými daty se v jazyce C++ často využívá standardní šablonová knihovna (STL – Standard Template Library). Součástí STL jsou datové kontejnery (např. vektor, mapa, fronta...), iterátory a algoritmy (umožňující řazení, hledání, kopírování...). Pokud by byly paralelní STL algoritmy poskytnuty uživatelům jako tzv. černé skříňky (black boxes), mohli by programátoři těžit z více-vláknového prostředí bez hlubší znalosti paralelizace.

Standard C++11 zavádí knihovnu Thread support library [6], která usnadňuje práci s vlákny. Až do verze 17 však standard C++ nepodporuje paralelní provádění STL algoritmů. Ve verzi C++17 je sice tato podpora teoreticky zavedena, v praxi však není, alespoň prozatím, tato funkcionalita do většiny kompilátorů implementována.

Při rešeršní práci nebyla nalezena efektivní in-place implementace paralelního quicksortu, která by využívala C++11 vláken a pouze standardní knihovny C++.

V této kapitole jsou popsána C++11 vlákna. Dále je popsán paralelizmus ve standardu C++17. Následuje přehled existujících implementací paralelního

quicksortu v jazyce C++.

2.1 C++11 vlákna

Do verze 11 C++ standard nepodporoval vytváření vícevláknových aplikací. Bylo nutné používat knihovny specifické pro daný systém, například POSIX (Portable Operating System Interface) vláknovou knihovnu, nebo jiné nestandardní knihovny, například knihovnu Boost.Thread. Přímé použití systémových vláken, jako je tomu v případě POSIX vláknové knihovny, je dnes považováno za nevhodné [7].

C++ od verze 11 zahrnuje knihovnu Thread support library, která usnadňuje vytváření paralelních programů. Tato knihovna zavádí podporu vláken, vzájemného vyloučení a podmíněných proměnných.

Vlákna umožňují spustit program na více jádrech zároveň. Vlákno je reprezentováno třídou. Do konstruktoru je předána funkce, která má být ve vláknu spuštěna, a její argumenty. Výpočet je spuštěn hned po konstrukci objektu. Na dokončení výpočtu vlákna se může, ale nemusí, čekat. Po ukončení provádění funkce předané do konstruktoru nelze vláknu přiřadit další práci.

Algoritmy vzájemného vyloučení zabraňují vláknům, aby současně přistupovala ke sdíleným zdrojům. Jejich použitím je možné zabránit synchronizačním chybám (data race). Ty vznikají například v případě, kdy více vláken přistupuje ke sdíleným prostředkům a alespoň jedno vlákno je mění.

Další důležitou součástí knihovny jsou podmíněné proměnné (condition variables). Podmíněné proměnné zjednodušují komunikaci mezi vlákny. Umožňují vláknům pasivně čekat na notifikaci z jiného vlákna. S jejich pomocí je možné vlákna synchronizovat.

Částečně knihovna Thread support library poskytuje abstrakci na vyšší úrovni. Tou je například konstrukt thread (vlákno), který spravuje jedno vlákno výpočtu. Thread je v praxi obvykle implementován obalením již existujících systémových knihoven pro správu vláken, například obalením posixových vláken (pthread).

Z části knihovna Thread support library obsahuje i nízkoúrovňové synchronizační konstrukty, kterými jsou například atomické proměnné, které nejsou dostupné při použití pouze posixových vláken.

2.2 Paralelizmus ve standardu C++17

Součástí standardu C++17 je standardizace technické specifikace paralelizmu [8]. Tato specifikace zavádí tzv. politiky provádění (execution policies). Jimi je možné specifikovat, zda má být operace prováděna sekvenčně, nebo může být prováděna paralelně, či paralelně s možností vektorizace. Ukázka použití těchto politik tak, jak je uvedeno v technické specifikaci, je v algoritmu 2.

Standard dále zavádí všeobecné formulace pro paralelní algoritmy a říká, které algoritmy by měly být paralelizovány. Jedná se o více než sedmdesát algoritmů, mezi které patří například algoritmy pro paralelní řazení, kopírování nebo vyhledávání.

V současné době je paralelizmus podporován pouze kompilátorem Intel C++ a částečně i kompilátorem MSVC (Microsoft Visual C++). Na nejběžněji používaných kompilátorech GCC (GNU Compiler Collection) a Clang však tato podpora chybí.

Algoritmus 2 Ukázka politik provádění

```
1: // standardní sekvenční sort
2: sort(v.begin(), v.end());
3:
4: // explicitní sekvenční sort
5: sort(sequential, v.begin(), v.end());
6:
7: // sort s povoleným paralelním prováděním
8: sort(par, v.begin(), v.end());
```

2.3 Existující implementace

V programovacím jazyce C++ existuje mnoho implementací paralelní verze algoritmu quicksort. Tato práce je zaměřena na návrh a efektivní implementaci paralelního in-place algoritmu quicksort, založenou na C++11 vláknech, ve více-vláknovém prostředí se sdílenou pamětí. Proto je zde kladen důraz právě na tyto typy řešení a nejsou zde analyzována řešení využívající například více procesorů.

Při rešerši byly hledány co možná nejpodobnější implementace. Nebyla nalezena žádná efektivní implementace in-place paralelního quicksortu založená na C++11 vláknech, která by nevyužívala nestandardní knihovny. Většina implementací využívá nestandardní specifikaci OpenMP (Open Multi-Processing) [9] nebo využívá externí knihovny, například Intel knihovnu paralelního STL [10]. Další nalezená řešení, která využívala pouze standardní knihovny a C++11 vlákna, nebyla in-place nebo byla implementována neefektivně.

V této části je poskytnut přehled nejvýznamnějších existujících implementací paralelního quicksortu v jazyce C++ určených pro více-vláknové prostředí se sdílenou pamětí.

2.3.1 Intel knihovna paralelního STL

Společnost Intel vytvořila knihovnu paralelního STL (Intel Parallel STL library) [10]. Tato knihovna implementuje algoritmy z C++ standardní knihovny.

Podporuje politiky provádění tak, jak jsou specifikovány v technické specifikaci paralelizmu, a následuje standardy C++17. Tato knihovna paralelního STL má tu nevýhodu, že pro její použití je nutné používat také rozsáhlou nestandardní knihovnu Threading Building Blocks (TBB).

Paralelní quicksort z této knihovny pracuje s iterátory náhodného přístupu (RAI – Random Access Iterator), které ukazují na začátek a konec řazené posloupnosti, a s funkcí, která slouží k porovnání řazených prvků. Před spuštěním samotného řazení je kontrolováno, jestli posloupnost již není seřazena. Díky tomu je tato implementace velmi rychlá pro již seřazené posloupnosti.

Řazená posloupnost je nejprve rozdělena na části. Ty jsou uloženy do front jednotlivých vláken. Počet částí je stanoven tak, že připadá na jedno výpočetní vlákno 64 částí. Části dále mohou být za běhu programu děleny na menší (rekurzivně na poloviny), pokud je to žádoucí. Každá část je řazena standardní funkcí pro sekvenční řazení. Vyvažování zátěže je docíleno použitím knihovny TBB. Pokud nějaké vlákno dokončí svoji práci, zatímco ostatní vlákna mají ještě hodně práce ve svých frontách, TBB přiřadí část práce zaneprázdněných vláken nečinnému vláknu.

Intel nabídl překladačům GCC a Clang svoji implementaci paralelních algoritmů [11]. Knihovna paralelního STL je tak v současnosti slučována do implementací standardních knihoven těchto překladačů libstdc++ a LLVM (Low Level Virtual Machine). Jsou odstraňovány nekonzistence se standardem. Zároveň je použití TBB abstrahováno interním API (Application Programming Interface), takže může být nahrazeno jiným řešením paralelních vláken. V době vytváření této práce však nebylo slučování dosud dokončeno. Problémy jsou i s licencí, která v současné době umožňuje pouze nekomerční užití. V budoucnu má však tato knihovna potenciál sloužit jako základ paralelní části knihovny GNU libstdc++.

Výsledky experimentálních měření [12] naznačují, že řazení s využitím Intel knihovny paralelního STL je více než dvakrát pomalejší než řazení s využitím paralelního módu knihovny libstdc++, který je popsán v části 2.3.3. I testy provedené v rámci této práce potvrzují, že tato knihovna nepatří v obecném případě mezi ty nejrychlejší. Může za to i relativně neoptimální implementace „kradení“ práce. Dle výzkumu Princetonské Univerzity [13] je kradení práce neoptimální pro velký počet jader. Tento výzkum ukázal, že při provádění experimentů na 32jádrovém systému je až 47 % výpočetního času zabráno režii na rozvrhování.

I některé další implementace paralelního řazení v prostředí více-vláknových aplikací se sdílenou pamětí využívají externí knihovnu Intel Threading Building Blocks. Vzhledem k nutnosti použití nestandardní knihovny a značné režii na rozvrhování u většího počtu jader však ani tyto implementace nejsou optimální.

2.3.2 Implementace založené na OpenMP

OpenMP je specifikace nestandardních rozšíření jazyků C, C++ a Fortran o konstrukty podporující více-vláknové programování. OpenMP obsahuje sadu direktiv, které ovlivňují chod programu. Direktivy jsou navrženy tak, že si program zachovává správné chování, i když je kompilátor nepodporuje. V tom případě je program spuštěn sekvenčně.

OpenMP se používá pro paralelní programování nad sdílenou pamětí. Ve zdrojovém kódu je nutné vyznačit direktivami bloky, které mají být prováděny paralelně. Po spuštění programu existuje jedno vlákno, které je označováno jako hlavní (master). Toto vlákno existuje po celou dobu běhu programu. Na začátku paralelního bloku vytvoří hlavní vlákno pomocí fork-join mechanismu příslušný počet paralelních vláken. Na konci paralelního bloku vlákna opět zaniknou. Z důvodu zmenšení režie procesu vytváření a zániku vláken je využíván fond vláken (thread pool).

Rozšíření OpenMP obsahuje podporu funkčního paralelismu. V paralelním regionu je možné použít konstrukt `task`. Ten slouží k vytvoření nové úlohy, která má být spuštěna. Po vytvoření je úloha vložena do tzv. fondu úloh (task pool). Všechna vlákna, která jsou součástí daného paralelního regionu, vybírají úlohy z tohoto fondu, dokud není prázdný. Konstrukt `task` je díky zapouzdření kódu i dat vhodný pro rekurzivní funkce.

Mnoho existujících implementací paralelního quicksortu používá právě rozšíření OpenMP. Důvodů je možné najít více. Před C++11 ani nebyla ve standardu C++ vlákna podporována. Z toho důvodu většina knihoven, které začaly vznikat před standardem C++11, pracuje s OpenMP a ne s C++11 vlákny. Rozšíření OpenMP je relativně snadné začít používat. Navíc na rozdíl od `std::thread` z C++11 samo implementuje další pomocné mechanizmy, například fond vláken nebo bariéru. OpenMP je zaměřeno hlavně na datový paralelizmus, ve kterém dosahuje vynikajících výsledků. Nicméně v paralelním quicksortu je spíše potřeba úlohový paralelizmus. Z toho důvodu je možné se domnívat, že by na implementaci paralelního quicksortu byla vhodnější C++11 vlákna, protože ta jsou více zaměřena na úlohový paralelizmus.

Cílem této práce je implementovat paralelní quicksort bez OpenMP, protože toto rozšíření není součástí standardu.

Nejvýznamnější implementace paralelního quicksortu, která využívá knihovnu OpenMP, je součástí paralelního módu knihovny libstdc++. Dále knihovnu OpenMP využívá například AQsort, který se od ostatních algoritmů liší tím, že dokáže řadit více polí najednou.

Vybraným zástupcem z mnoha méně významných implementací paralelního quicksortu využívající OpenMP je řazení z knihovny Sorter Threaded [14]. Toto řazení navíc používá pomocnou paměť, která je lineární vzhledem k délce řazené posloupnosti. Oproti běžné implementaci umožňuje díky pomocné paměti rozdělení posloupnosti na libovolný počet částí. Ve výchozím nastavení rozděluje řazenou posloupnost tak, aby každé vlákno dostalo při-

děleno osm úloh. Samotné rozdělování posloupnosti o velikosti n na c částí probíhá v čase $\mathcal{O}(n \cdot log(c))$. Rozdělení na c částí probíhá na základě $c-1$ pivotů. V této implementaci jsou pivoti voleni jako prvních $c-1$ prvků posloupnosti. To může být značně neefektivní, hlavně u částečně seřazených posloupností. Přiřazování řazených částí je rozvrhováno dynamicky za běhu programu. Podobně jako mnoho jiných implementací paralelního quicksortu není ani tato implementace zcela optimální, především kvůli sekvenčnímu rozdělování. Tato implementace je zde uvedena především kvůli porovnání s efektivnějšími implementacemi.

2.3.3 Paralelní mód libstdc++

Implementace GNU standardní C++ knihovny se nazývá libstdc++. Knihovna libstdc++ obsahuje experimentální paralelní mód [5] pro některé paralelní algoritmy. S použitím tohoto módu je možné jednoduše změnit existující sekvenční kód tak, aby běžel paralelně na více vláknech. Použití paralelního módu je nezávislé na platformě. Implementace paralelního módu byla původně součástí MCSTL (Multi-Core Standard Template Library) [15]. Následně byla integrována do implementace GNU C++ překladače.

Paralelní mód umožňuje například paralelní řazení, spojování, rozdělování nebo paralelní prefixové součty. Mód využívá rozšíření OpenMP. Autoři paralelního módu knihovny libstdc++ zvolili toto řešení, protože OpenMP podporuje fork-join paralelizmus a také protože mohli na OpenMP ponechat nízkoúrovňové záležitosti, jako je například zjišťování maximálního počtu současně běžících vláken.

Autoři uvádí, že velikost paralelního kódu algoritmů je až čtyřnásobná oproti sekvenčnímu. Z toho důvodu je programátorům doporučeno vybírat vhodné paralelní algoritmy již při kompilaci, ne až za běhu programu. Za další problém do budoucna je autory považováno detekování vhodného bodu pro automatické přepnutí z provádění paralelního algoritmu na provádění sekvenční.

Knihovna libstdc++ implementuje také paralelní řazení. V knihovně je implementováno více algoritmů – vícecestný merge sort (ve více variantách), quicksort a vyvážený quicksort. Pro účely rešerše jsou zajímavé obě verze algoritmu quicksort. Obě verze přijímají ukazatel na počátek a konec řazené posloupnosti a funkci pro porovnávání prvků. Počet vláken, která mají na řazení pracovat, může být specifikován. Pokud počet není specifikován, použije se počet vláken paralelního bloku OpenMP, ve kterém bylo řazení spuštěno.

Při použití nevyváženého quicksortu je řazená posloupnost na počátku rozdělena na dvě části. Pivot je zvolen jako medián hodnot vybraných rovnoměrně z celé řazené posloupnosti. Ve výchozím nastavení je počet vybraných hodnot sto. Posloupnost je rozdělována paralelně všemi dostupnými vlákny. Rozdělování je efektivní, založené na vytváření bloků. Po rozdělení je řazení rekurzivně zavoláno na obě části. Každá část je následně řešena polovičním

počtem vláken, nezávisle na tom, jak jsou obě poloviny velké. Pokud počet vláken, která mají řadit posloupnost, klesne na jedno vlákno, je posloupnost dále řazena sekvenčním standardním řadícím algoritmem.

V paralelním quicksortu není hlídána hloubka rekurze, která je obvykle sledována v sekvenčním quicksortu. Jejím využitím je zabránění případnému nejhoršímu případu, kdy by řazení pomocí quicksortu mělo složitost $\mathcal{O}(n^2)$. Hlídání hloubky rekurze zde není nutné. Kvůli dělení počtu zúčastněných vláken vždy na polovinu může být hloubka paralelního quicksortu maximálně logaritmická vzhledem k počtu vláken. Poté dochází k přepnutí na sekvenční quicksort. V sekvenčním quicksortu je již omezena hloubka rekurze.

Vyvážený quicksort se liší tím, že dynamicky vyvažuje zátěž jednotlivých vláken. Je podobný algoritmu, který navrhl P. Tsigas a Y. Zhang [16]. Každé vlákno si udržuje ukazatele na začátek a konec podposloupnosti, na které pracuje. Každému vláknu navíc přísluší fronta s částmi, které jsou určeny k dalšímu zpracovávání. Tato fronta umožňuje „kradení“ částí jinými vlákny, neboť vlákna mají přístup k frontám jiných vláken. Každá fronta má velikost $p \cdot log(n)$, kde p je počet vláken, která se paralelního řazení účastní a n je počet prvků řazené posloupnosti. Fronta v sobě nikdy nemůže mít více částí, protože největší část má nejvýše n prvků. Počet prvků každé další vložené části je nejvýše poloviční oproti počtu prvků předchozí vložené části.

Vyvážený quicksort se od nevyváženého liší výběrem pivota. Protože zde není tolik důležité rozdělení posloupnosti na dvě rovnoměrné části, pivot je volen jako medián z prvního, prostředního a posledního prvku řazené posloupnosti. Samotné rozdělování probíhá stejně jako u nevyváženého quicksortu.

Vlákna, která se mají dále zúčastňovat řazení podposloupností, jsou zde rozdělena ne na poloviny, jak tomu je u nevyváženého quicksortu, ale poměrově vzhledem k velikostem obou částí. Vzhledem k výběru pivota z pouhých tří prvků může být velikost obou částí dost rozdílná. Pokud počet vláken, která se účastní řazení, klesne na jedno, nebo velikost řazené posloupnosti klesne pod určitou mez, začíná lokální řazení s pomáháním.

Pokud je řazená posloupnost velká, je posloupnost sekvenčně rozdělena na menší za použití náhodného pivota. Větší část je uložena na zásobník, menší část je řazena dále iterativně. Díky tomu je hloubka rekurze maximálně logaritmická. Pokud velikost řazené posloupnosti klesne pod určitou mez, je posloupnost seřazena sekvenčně bez rozdělování na menší části. Ve výchozím nastavení je tato mez stanovena na sto prvků. Po seřazení se vlákno pokusí získat další posloupnost k seřazení ze své fronty. Při vybírání práce ze své fronty preferuje malé kousky práce. Pokud již ve své frontě žádnou práci nemá, vybere náhodně vlákno, kterému se pokusí pomoct. Pokusí se ukrást práci z jeho fronty. Snaží se ukrást co největší kus práce, tedy vybírá práci z opačného konce fronty než při vybírání ze své fronty. Vlákna se stále aktivně snaží krást práci jiným vláknům, i když nemusí být žádná práce k dispozici.

Vyvážený quicksort potřebuje ke svému běhu logaritmické množství paměti navíc, oproti nevyváženému quicksortu. V průběhu testování dosahoval vy-

vážený quicksort v průměrném případě lepších výsledků než quicksort nevyvážený. Naopak dosahoval horších výsledků na opačně seřazených datech.

Hlavní nevýhodou výše popsaných implementací z knihovny libstdc++ je, že ke svému běhu potřebují již zmíněnou nestandardní specifikaci OpenMP. Celkově jsou obě tyto implementace velmi rychlé.

2.3.4 AQsort

Další verzí paralelního algoritmu quicksort je AQsort [4]. Od běžných verzí paralelního quicksortu se liší tím, že umožňuje řadit více polí najednou. Jeho implementace uvedená ve stejném článku je v jazyce C++ a využívá rozšíření OpenMP.

Od ostatních implementací se AQsort liší tím, že nemá prakticky žádné informace o řazených datech. Neví, kolik prvků řadí, jakého jsou typu, ani nemůže k řazeným prvkům přistupovat. Prvky, které se nachází na stejné pozici ve všech řazených polích, se nazývají *multielementy*. Parametry AQsortu jsou, kromě jiných, reference na dvě funkce. První z nich je funkce pro porovnávání. Té jsou pouze předány pozice multielementů, které mají být porovnány. Funkce vrací booleovskou hodnotu indikující, jestli je multielement na první pozici menší než multielement na druhé pozici. Druhou funkcí je funkce pro prohození. Ta provádí prohození dvou multielementů na zadaných pozicích. Implementace funkce pro porovnávání a funkce pro prohození je ponechána na uživateli AQsortu.

V algoritmu AQsort je kontrolována i v paralelní části hloubka zanoření rekurzivního volání. Rozdělování řazené posloupnosti na dvě části je prováděno paralelně v blocích, podobně jako v paralelním módu knihovny libstdc++. Velikost bloků, používaných v paralelním rozdělování algoritmu AQsort, je ve výchozím nastavení stanovena na 1024. Při výběru pivota je hledán medián tří mediánů ze tří prvků.

2.3.5 Microsoft Visual C++

Microsoft Visual C++ (MSVC) je vývojové prostředí Microsoftu pro jazyky C a C++. Obsahuje i vlastní MSVC kompilátor. MSVC často pracuje s knihovnami závislými na platformě, například použitelnými pouze na systému Windows.

Součástí Microsoft Visual C++ je vlastní knihovna podporující vytváření paralelních algoritmů nazývaná Parallel Patterns Library (PPL) [10]. Tato knihovna obsahuje i implementaci paralelního řazení. Jedná se o nestabilní inplace implementaci. Obsahuje verze jak pro rozdělování posloupnosti na dvě části, tak pro rozdělování na tři části. Ve druhé verzi se navíc objevuje část s prvky, které jsou rovny pivotovi.

V průběhu paralelního quicksortu je nejprve volen pivot. U menších posloupností je pivot vybírán jako medián ze tří prvků. U větších posloupností je

pivotem medián tří mediánů ze tří prvků. Samotné rozdělování posloupnosti na dvě menší není paralelizováno. Implementace je naivní, stejná jako u sekvenční verze quicksortu. Z toho důvodu je možné, že tato implementace bude hůře škálovatelná než konkurenční implementace.

Po rozdělení na dvě části je z druhé části vytvořena nová úloha, která je spuštěna. První část je následně seřazena rekurzivním voláním řadící funkce. Po navrácení z rekurzivního volání se čeká na dokončení řazení druhé části.

V případě dosažení maximální hloubky rekurze, malé velikosti řazené posloupnosti nebo dosažení maximálního počtu úloh je paralelní quicksort přepnut na sekvenční standardní řadící algoritmus. Maximální hloubka rekurze je nastavena konstantně na 64. Výchozí hodnota nastavení celkového počtu úloh, které mohou být za běhu programu vytvořeny, je stanovena na $p \cdot 1024$, kde p je počet dostupných vláken.

Knihovna PPL je částečně dostupná na různých platformách. Část s paralelními algoritmy je však dostupná pouze na platformě Windows. Vzhledem k tomu, že všechna měření v testovací fázi probíhala na serveru s platformou Linux, nebylo možné tuto implementaci otestovat za stejných podmínek. Podle výsledků měření B. Neala [17] trvalo seřazení milionu hodnot typu double v sekvenční verzi přibližně 75 sekund. Při paralelizaci na systému s 18 jádry a 36 vlákny bylo dosaženo seřazení stejných dat za přibližně 19 sekund. Navíc stejný zdroj udává, že paralelní verze byla rychlejší než sekvenční pro sekvence o alespoň tisíci prvcích.

Nevýhodou knihovny PPL je kromě obtížné přenositelnosti na jiné platformy než Windows také absence open-source licence. To dělá knihovnu pro mnohé projekty v praxi nepoužitelnou.

2.3.6 Implementace založené na C++11 vláknech

Cílem této práce je vytvořit efektivní implementaci paralelního quicksortu na základě C++11 vláken. Proto byly při rešerši hledány implementace, které by C++11 vlákna využívaly. V rámci rešerše se nepodařilo najít implementaci paralelního quicksortu, která by využívala C++11 vlákna a zároveň byla efektivní a in-place.

Jedna z nalezených relevantních implementací je součástí knihovny Parallel file quicksort [18]. Tato implementace využívá vlákna z knihovny thread, je in-place, ale je přibližně sedmkrát pomalejší než konkurenční implementace. Pivot je volen jako medián ze tří prvků. Implementace rozdělování posloupnosti je pouze naivně sekvenční. I ostatní nalezené knihovny pro in-place paralelní řazení byly podobně málo efektivní.

Další zajímavou implementací paralelního quicksortu je parasort [19]. Tato implementace dosahuje pro nepříliš velká vstupní data (v provedených testech přibližně do 1 miliardy čísel) výborných výsledků. Doba běhu programu je často přibližně poloviční ve srovnání s výše zmíněnými algoritmy. Avšak tato implementace vyžaduje v rozdělování pomocnou paměť o velikosti řazené po-

sloupnosti. Řazenou posloupnost při tom třídí do tolika skupin, kolik je vláken. Tím se z quicksortu stává out-of-place algoritmus. Tato práce je zaměřena na in-place algoritmy. Tento přístup však stojí za povšimnutí, pokud by byla přípustná out-of-place varianta.

Kapitola 3

Rozdělování

Rozdělování (partitioning) posloupnosti prvků na části je klíčovou součástí mnoha algoritmů. Využívá se například u algoritmů quicksort nebo quickselect. Rozdělování je prováděno vzhledem ke zvolené hodnotě, která se nazývá pivot.

Cílem je přeuspořádání vstupní posloupnost tak, že pro nějakou rozdělovací pozici s platí, že prvky nalevo od této pozice nejsou větší než pivot a zároveň prvky napravo od této pozice nejsou menší než pivot.

Jak je uvedeno v kapitole 1.3.2 v implementaci paralelních řadících algoritmů je vhodné, aby i rozdělování probíhalo paralelně, protože sekvenční rozdělování by výrazně zamezovalo škálovatelnosti paralelního algoritmu.

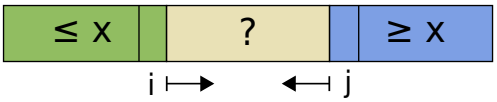
V této kapitole jsou popsány algoritmy pro sekvenční rozdělování posloupnosti. Následně jsou představeny out-of-place a in-place varianty paralelního rozdělování. V poslední části je podrobně představeno rozdělování tak, jak je implementováno v této práci.

3.1 Sekvenční rozdělování

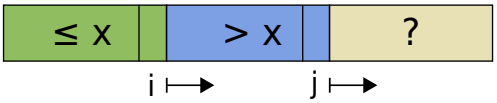
V případě sekvenčního quicksortu je obvykle používán jeden ze dvou způsobů rozdělování [20].

První způsob rozdělování spočívá v tom, že rozdělovaná posloupnost je procházena dvěma iterátory, jedním zleva a druhým zprava. Levým iterátorem je hledán prvek, jehož hodnota je větší než pivot. Pravým iterátorem je hledán prvek, který je menší než pivot. Pokud oba iterátory naleznou tyto prvky, prvky jsou prohozeny a iterátory pokračují v procházení posloupnosti, dokud se nepotkají. Tento způsob se někdy nazývá Hoare rozdělování. Princip Hoare rozdělování je znázorněn na obrázku 3.1.

Druhý způsob spočívá v tom, že levý iterátor začíná na prvním prvku a pravý iterátor začíná na druhém prvku posloupnosti. Pravý iterátor se pohybuje směrem doprava. Pokud ukazuje pravý iterátor na prvek, jehož hodnota je menší než pivot, jsou hodnoty, na které iterátory ukazují, prohozeny. Následně



Obrázek 3.1: Hoare sekvenční rozdělování posloupnosti, pivot je označen x.



Obrázek 3.2: Lomuto sekvenční rozdělování posloupnosti, pivot je označen x.

se levý iterátor posune o jeden prvek doprava. Pravý iterátor pokračuje v pohybu směrem doprava, dokud nedosáhne konce posloupnosti. Tento způsob se někdy nazývá Lomuto rozdělování. Princip Lomuto rozdělování je znázorněn na obrázku 3.2.

Lomuto rozdělování dosahuje horších výsledků na posloupnostech, které obsahují hodně duplicitních hodnot. I v průměrném případě provede Lomuto rozdělování třikrát více prohazování než Hoare rozdělování [20]. Oba způsoby přistupují do paměti sekvenčně, takže využití cache pamětí je prakticky optimální. Obecnou výhodou Lomutova rozdělování je fakt, že může být použito na jednosměrně zřetězeném spojovém seznamu. Hoare rozdělování vyžaduje obousměrné iterátory. V sekvenčních řadících algoritmech pro všeobecné použití se obvykle používá Hoare rozdělování.

3.2 Paralelní out-of-place rozdělování

Paralelní rozdělování je složitější než sekvenční. Relativně jednodušší implementace paralelního rozdělování vyžaduje pomocné místo o velikosti rozdělované posloupnosti. Tento přístup není in-place, ale je v praxi u implementací paralelního quicksortu často používán, protože je velmi rychlý.

Posloupnost o velikosti n je rozdělována p vlákny. Každé vlákno zpracovává stejně velkou souvislou část o velikosti n/p. Je zvolena prvek, který má být použit jako pivot, například mediánem ze tří vybraných prvků. Dále jsou vytvořena dvě pomocná pole o velikosti p, nazývána *mensi* a *vetsi*. Každé vlákno projde sekvenčně svoji část. Při tom do pomocných polí zaznamená počet prvků, které jsou menší než pivot a počet prvků, které jsou větší než pivot. Následně je na polích *mensi* a *vetsi* proveden prefixový součet. Je vytvořeno výstupní pole o velikosti n. Do něj jednotlivá vlákna vkládají prvky ze svých částí. Indexy, na které mají prvky vkládat, jsou určeny z pomocných polí *mesni* a *vetsi*.

3.3 Paralelní in-place rozdělování

Tato práce má za cíl vytvořit in-place implementaci paralelního quicksortu. Z toho důvodu musí být i rozdělování in-place. Existuje několik algoritmů pro paralelní in-place rozdělování. Mezi nejvýznamnější patří základní algoritmus od autorů Francis a Pannan [21], načti-a-přidej (F&A) algoritmus od autorů Tsigas a Zang [16] a jemu podobný algoritmus ze standardní šablonové vícejádrové knihovny (MCSTL) [15], který je používán i v knihovně libstdc++.

Společným znakem výše zmíněných algoritmů je, že provedou více operací než sekvenční rozdělování. V sekvenční verzi je možné rozdělit posloupnost o velikost n za použití n porovnání a m prohození, kde m je počet prvků větších než pivot, jejichž pozice na vstupu je menší než pozice pivota na výstupu. Leonor Frias a Jordi Petit přišli ve svém článku [22] s modifikací těchto algoritmů, která vede k dosažení minimálního počtu porovnání.

Všechny níže uvedené algoritmy paralelního rozdělování se skládají ze tří hlavních fází.

1. **Přípravná fáze** – pro každé vlákno je sekvenčně připravena práce.
2. **Hlavní fáze** – zde je paralelně provedena většina rozdělování.
3. **Úklidová (cleanup) fáze** – dosud nerozdělené menší kousky posloupnosti jsou zpracovány.

3.3.1 Krokové a blokové paralelní rozdělování

Krokové (strided) paralelní rozdělování navrhli Francis a Pannan [21]. Průběh algoritmu je znázorněn na obrázku 3.3.

V přípravné fázi je posloupnost rozdělena na p částí, kde p je počet vláken, která se paralelního rozdělování účastní. Jednomu vláknu jsou přiřazovány prvky s krokem p, tedy i-té vlákno bude rozdělovat prvky na pozicích $i + 0$, $i + p$, $i + 2p$ ...

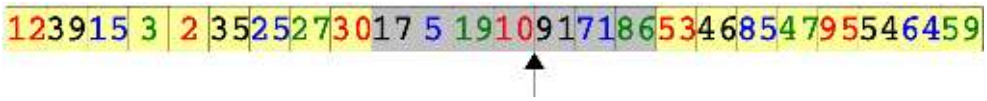
V hlavní fázi každé vlákno rozděluje sekvenčně své prvky. Po skončení je z každého vlákna navrácena rozdělovací pozici rozdělované podposloupnosti v_i.

V úklidové fázi musí být sekvenčně rozděleny prvky, které se nachází v intervalu, který není korektně rozdělen. Nechť v_{min} označuje nejnižší rozdělovací pozici ze všech navrácených pozic a v_{max} označuje nejvyšší rozdělovací pozici. Všechny prvky na pozicích menších než v_{min} a větších než v_{max} jsou již umístěny korektně vzhledem k pivotovi. Všechny prvky mezi těmito hodnotami mohou být chybně umístěny vzhledem k pivotovi, proto je interval (v_{min}, v_{max}) rozdělen sekvenčně.

Z hlediska časové složitosti trvá hlavní fáze $\Theta(n/p)$, kde n je délka rozdělované posloupnosti a p je počet využívaných vláken. Složitost poslední fáze je v průměrném případě pouze $\mathcal{O}(1)$, ale v nejhorším případě může být i $\Theta(n)$.

(a) Přípravná fáze.

(b) Hlavní fáze.



(c) Úklidová fáze.

Obrázek 3.3: Průběh krokového rozdělování. Pivot má hodnotu 40 a dělení probíhá čtyřmi vlákny. Obrázek je převzat od L. Friase [23].

Další nevýhodou tohoto algoritmu je špatná lokalita mezipaměti, protože všechna vlákna přistupují do posloupnosti prakticky po celé její délce.

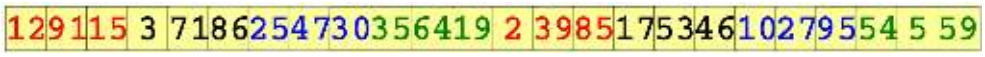
Úpravou krokového algoritmu na blokový je možné výrazně zlepšit lokalitu mezipamětí. Blokový algoritmus se od předchozího se liší tím, že vláknům jsou přiřazovány bloky b po sobě jdoucích prvků s krokem $b \cdot p$. Průběh blokového algoritmu je znázorněn na obr. 3.4.

3.3.2 F&A paralelní rozdělování

Dalším vylepšením výše popsaného algoritmu je varianta, kdy vlákna získávají prvky z obou konců posloupnosti podle potřeby. Není staticky na začátku přiděleno, které prvky bude vlákno zpracovávat. Prvky jsou získávány z obou konců v blocích. Pokud by byly prvky získávány po jednom, bylo by třeba příliš mnoho synchronizace mezi vlákny. Algoritmus se jmenuje F&A, protože používá tzv. fetch-and-add (načti-a-přidej) instrukce. Tyto instrukce umožňují atomicky zvýšit proměnnou a vrátit její původní hodnotu.

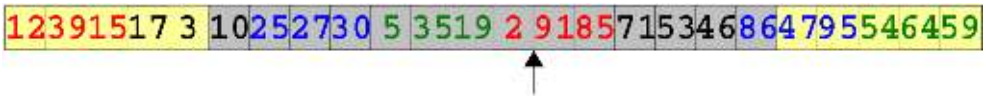
V přípravné fázi si každé vlákno přivlastní jeden blok zleva a jeden blok zprava. Právě při získávání bloků jsou využity fetch-and-add instrukce, aby byl každý prvek posloupnosti zpracováván právě jedním vláknem.

V hlavní fázi každé vlákno provádí sekvenční rozdělování prvků z levého a pravého bloku. Při dosažení hranice některého bloku je daný blok označen jako „neutralizovaný“ a vlákno si přivlastní další blok z té strany, ze které pochází právě neutralizovaný blok.



(a) Přípravná fáze.

(b) Hlavní fáze.



(c) Úklidová fáze.

Obrázek 3.4: Průběh blokového rozdělování. Pivot má hodnotu 40, dělení probíhá čtyřmi vlákny a velikost bloku je 3. Obrázek je převzat od L. Friase [23].

V úklidové fázi zbývá nejvýše p bloků, které ještě nejsou zcela rozděleny, tedy nejsou neutralizovány. Ve verzi Tsigas a Zang je tato fáze prováděna sekvenčně. Dokud to je možné, je brán jeden nezneutralizovaný blok zleva a jeden zprava a je nad nimi prováděno sekvenční rozdělování. Poté jsou nerozdělené bloky přesunuty do středu posloupnosti mezi neutralizované bloky. Nakonec je sekvenčně rozdělen interval tvořený dosud nerozdělenými bloky.

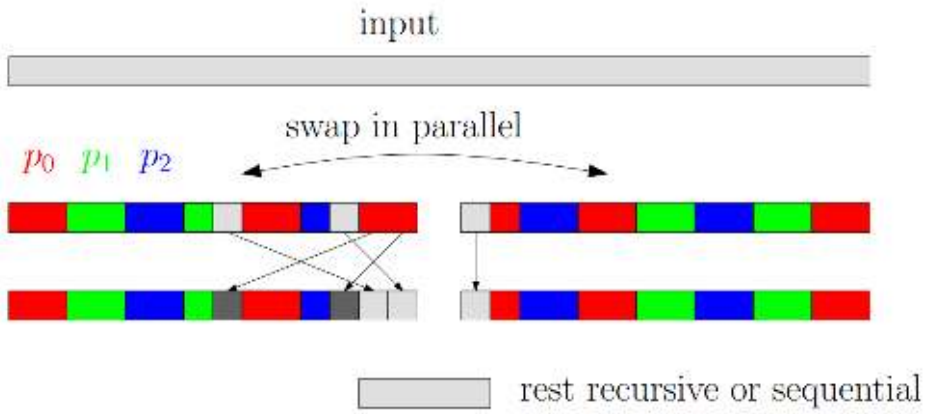
Ve verzi MCSTL je úklidová fáze prováděna paralelně. Všechny nezneutralizované bloky jsou přesunuty do středu posloupnosti. Následně je rekurzivně aplikováno paralelní rozdělování s poloviční počtem vláken. Pokud zbývá k rozdělování pouze jeden blok, nebo je dostupné pouze jedno vlákno, je zbývající část rozdělena sekvenčně.

Princip F&A paralelní rozdělování je znázorněn na obrázku 3.5. Z hlediska časové složitosti trvá hlavní fáze $\Theta(n/p)$. Úklidová fáze trvá ve verzi Tsigas a Zang $\mathcal{O}(b \cdot p)$, ve verzi MCSTL $\Theta(b \cdot \log(p))$.

3.3.3 Možné vylepšení úklidové fáze

Ve výše popsaných algoritmech paralelního rozdělení jsou v úklidové fázi přesunuty do středu všechny bloky, které nebyly neutralizovány. Ty jsou následně zpracovány bez ohledu na to, že během hlavní fáze byla v jejich části provedena neutralizace.

Úklidovou fázi je možné vylepšit tak, že neprovádí nepotřebná porovnání a prvky jsou v jejím průběhu prohazovány paralelně. Vylepšená úklidová fáze

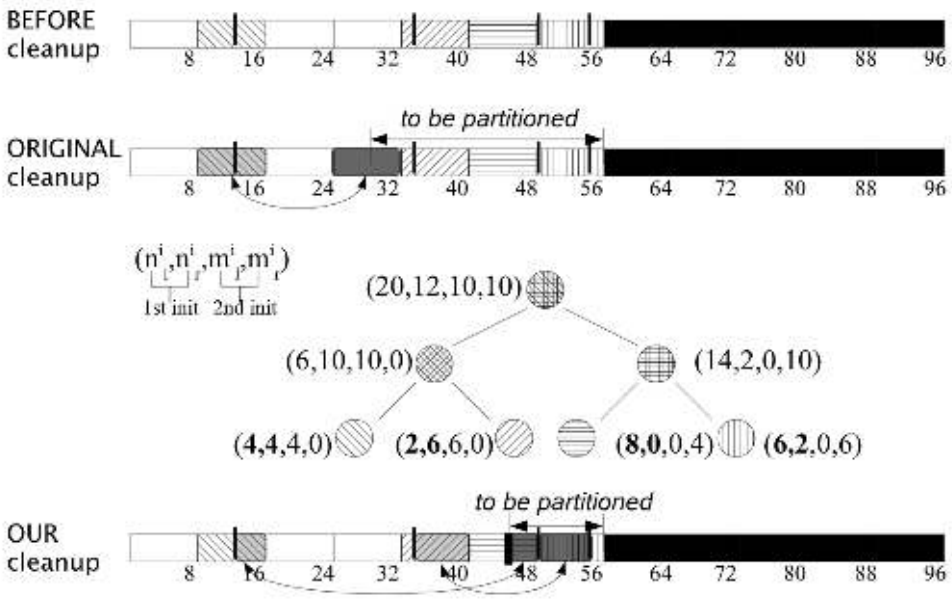


Obrázek 3.5: F&A paralelní rozdělování posloupnosti. Obrázek je převzat z prezentace J. Singlera [24].

pro algoritmus F&A je popsána L. Friasem [22] následovně: Celkový počet chybně umístěných prvků (misplaced elements) je označen m. Proměnnou M je označen počet chybně umístěných bloků. Pravé a levé bloky jsou zpracovávány zvlášť, ale jejich zpracovávání je analogické. Termínem *hranice* je označováno místo v bloku, které rozděluje zpracovanou a nezpracovanou část. Chybně umístěnými prvky jsou nazývány nezpracované prvky, které nejsou v prostředním intervalu, a zpracované prvky, které jsou v prostředním intervalu. Počet chybně umístěných bloků M je nejvýše $2p$, kde p je počet vláken. Může existovat maximálně p nezneutralizovaných bloků, které mohou být všechny chybně umístěné.

Následně je vytvořen binární strom s M listy, který je sdílený všemi vlákny. Listy obsahují informace o blocích. Každý list obsahuje čtyři údaje. Obsahuje počet chybně umístěných prvků nalevo a napravo od hranice a celkový počet prvků nalevo a napravo od hranice. Vnitřní uzly obsahují akumulované informace z jejich potomků. Tuto datovou strukturu je možné využít k rozhodnutí, které prvky mají být prohozeny bez provádění dalších porovnání. Všechny chybně umístěné prvky, které je třeba prohodit, jsou rovnoměrně rozděleny mezi jednotlivá vlákna. Porovnání originální a vylepšené verze úklidové fáze je znázorněno na obrázku 3.6.

Teoreticky toto vylepšení zrychluje, za předpokladu $p \leq b$, úklidovou fázi z $\Theta(b \log(p))$ na $\Theta(\log^2(p) + b)$. V praxi však toto vylepšení nedosahuje lepších výsledků, protože počet chybně umístěných prvků je relativně malý. Limitaci reálného využití tohoto vylepšení připouští ve výše zmíněném článku i jeho autoři. Ani v průběhu implementace nebyl pozorován jeho přínos, proto se vylepšená úklidová fáze ve finální implementaci nevyskytuje.



Obrázek 3.6: Porovnání originální a vylepšené úklidové fáze paralelního rozdělování. Obrázek je převzat z článku L. Friase [22].

3.4 Implementace rozdělování

Ve výsledné implementaci in-place paralelního quicksortu založené na C++11 vláknech je rozdělování důležitou součástí algoritmu. Na nejvyšších úrovních rekurze musí být prováděno paralelně, aby bylo řazení dobře škálovatelné. Při vývoji implementace rozdělování bylo experimentálně zjištěno, že nejlepších výsledků dosahuje implementace založená na F&A rozdělování ve verzi MCSTL, která je nakonec ve výsledné implementaci použita. Samotné rozdělování je implementováno samostatně, nezávisle na algoritmu řazení. Je tedy možné ho využít i v jiných algoritmech.

Implementace rozdělování z MCSTL je založena na rozšíření OpenMP, zde vytvořená implementace je založena na C++11 vláknech a příslušných synchronizačních technikách. Princip implementovaného rozdělování (při použití tří vláken) je znázorněn na obrázcích 3.7 až 3.12.

Na vstupní posloupnost je pohlíženo jako na posloupnost datových bloků o velikosti b. Jednotlivá vlákna získávají bloky, které dále zpracovávají. Bloky je možné získávat buď zleva, nebo zprava.

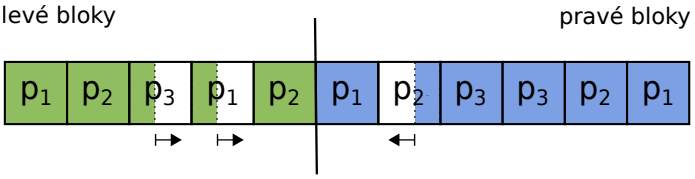
Každé vlákno na počátku získá jeden blok zleva a jeden blok zprava. Počáteční přiřazení bloků vláknům je znázorněno na obrázku 3.7. V rámci jednoho vlákna je nad těmito bloky prováděna neutralizace. Ta odpovídá sekvenčnímu Hoare rozdělování. Spočívá v prohazování prvků mezi levým a pravým blokem. Prvky v blocích jsou sekvenčně procházeny. Levý blok je procházen zleva doprava, pravý blok je procházen zprava doleva. Pokud je

Obrázek 3.7: Počáteční přiřazení bloků vláknům.

v levém bloku nalezen prvek, který je větší než pivot, a v pravém bloku je nalezen prvek, který je menší než pivot, jsou tyto prvky prohozeny.

V případě, že při procházení bloku je dosaženo konce bloku, je blok považován za neutralizovaný. Levé neutralizované bloky obsahují pouze prvky menší než je pivot (nebo jemu rovny). Pravé neutralizované bloky obsahují pouze prvky větší než je pivot (nebo jemu rovny). Pokud byl neutralizován levý blok, vlákno se pokusí získat další levý blok. Obdobně se snaží získat další pravý blok při neutralizaci bloku pravého. Neutralizace pokračuje, dokud je možné získávat další bloky.

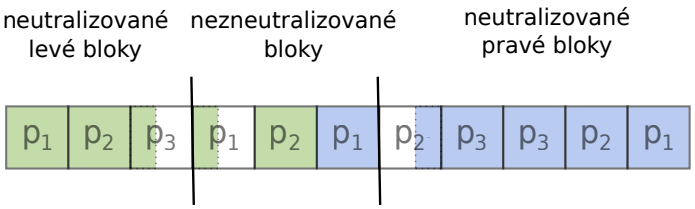
Pokud již není možné získávat další bloky, může mít vlákno oba své poslední zpracovávané bloky neutralizované, nebo může být nezneutralizovaný levý blok, nebo může být nezneutralizovaný pravý blok. Není možné, aby oba bloky byly nezneutralizované. Tedy celkový počet nezneutralizovaných bloků je nejvýše roven počtu vláken. Situace, kdy již není možné získávat další bloky, a proto končí paralelní neutralizace, je znázorněna na obrázku 3.8.



Obrázek 3.8: Konec paralelní neutralizace. Nelze získávat další bloky pro neutralizaci.

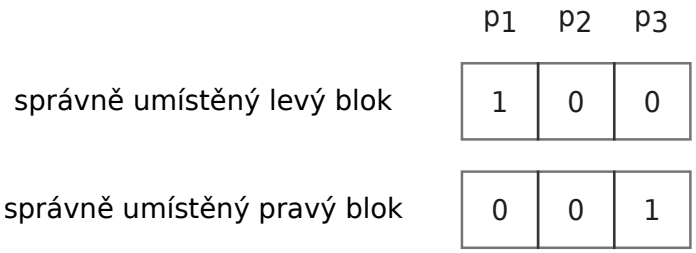
Dále je napočítáno, kolik levých bloků není neutralizováno a kolik pravých bloků není neutralizováno. V tomto bodě je nutné synchronizovat všechna vlákna pomocí bariéry, aby všechna vlákna měla úplnou informaci o počtu nezneutralizovaných levých a pravých bloků. Dále vlákna pokračují opět paralelně.

Následně jsou na základě počtů nezneutralizovaných bloků vypočítány hranice intervalu, který se nachází mezi neutralizovanými levými bloky a neutralizovanými pravými bloky. V tomto intervalu se mají nacházet bloky, které nejsou neutralizované. Rozdělení posloupnosti na první oblast pro neutralizované levé bloky, druhou oblast pro nezneutralizované bloky a třetí oblast pro neutralizované pravé bloky je znázorněno na obrázku 3.9.



Obrázek 3.9: Posloupnost je rozdělena na tři oblasti. V první mají být neutralizované levé bloky. Ve druhé mají být dosud nezneutralizované bloky. Ve třetí oblasti mají být neutralizované pravé bloky.

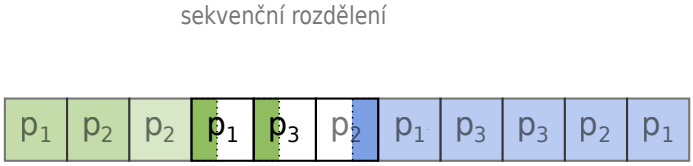
Každé vlákno pro své dva bloky, které naposledy zpracovávalo, poznamená, jestli se nachází na správném místě. Správné místo pro neutralizovaný blok je v krajních, neutralizovaných oblastech. Správné místo pro nezneutralizovaný blok je ve vnitřním intervalu. Správnost umístění bloků je znázorněna na obrázku 3.10.Po zjišťování správnosti umístění bloků je opět nutná synchronizace vláken pomocí bariéry, aby měla všechna vlákna kompletní informaci o tom, které bloky jsou na správných místech.



Obrázek 3.10: Zaznamenání správnosti umístění bloků. Jednička značí, že je daný blok příslušného vlákna umístěn správně. Nula značí, že je blok umístěn špatně.

Poté všechna vlákna opět běží paralelně. Pokud je některý nezneutralizovaný blok vlákna umístěn špatně, je pro něj nalezen neutralizovaný blok ze stejné strany, který je také špatně umístěn. Tyto dva bloky jsou prohozeny, čímž se oba stanou dobře umístěnými. Prohazování bloků je znázorněno na obrázku 3.11.

Po dokončení prohazování jsou všechny bloky správně umístěné. Tedy v první části jsou levé neutralizované bloky, ve druhé části jsou nezneutralizované bloky a ve třetí části jsou pravé neutralizované bloky. Tím končí paralelní sekce a následné rozdělování nezneutralizovaných bloků probíhá sekvenčně Hoare rozdělováním. Závěrečná úklidová fáze se sekvenčním rozdělováním je znázorněna na obrázku 3.12.



Obrázek 3.12: V poslední, úklidové, fázi jsou dosud nezneutralizované bloky sekvenčně rozděleny.

Sekvenční rozdělování nezneutralizovaných bloků při vhodné velikosti bloků příliš nezpomaluje běh programu, protože se jedná pouze o zanedbatelné množství prvků. Nezneutralizovaných bloků může být maximálně pouze tolik, kolik je vláken. Za předpokladu, že velikost bloku krát počet vláken je řádově menší než celkový počet prvků, není nutné tuto část paralelizovat. Po dokončení této části je rozdělování dokončeno a návratovou hodnotou rozdělování je pozice, která odděluje prvky menší než pivot a prvky větší než pivot.

Při rozdělování je nutné zvolit správnou velikost bloku. Menší bloky vyžadují více synchronizace mezi vlákny. Větší bloky vyžadují méně synchronizace, ale na druhou stranu je práce hůře rozdělitelná mezi jednotlivá vlákna. Pokud by bloky byly příliš velké, takže by řádově připadalo pouze pár bloků na vlákno, byla by posloupnost řazená v závěrečné uklízecí fázi příliš velká a z paralelního rozdělování by se stalo sekvenční rozdělování. Blok by měl být tak velký, aby se vešel do L1 mezipaměti. Po sérii experimentů během vývoje algoritmu byla zvolena jako výchozí hodnota velikosti bloku šest tisíc prvků.

Kapitola 4

C++11 paralelní quicksort

Během rešeršní práce nebyla nalezena efektivní implementace paralelního algoritmu quicksort, která by byla založena na C++11 vláknech a využívala pouze standardních knihoven. Proto byla navržena a implementována efektivní verze paralelního algoritmu quicksort, která je založena na C++11 vláknech a příslušných synchronizačních technikách.

Důležitou součástí paralelních algoritmů je rozdělování práce mezi jádra procesoru tak, aby všechna obdržela stejné množství práce. Ideálně by nemělo docházet k tomu, aby nějaké jádro nečinně čekalo, zatímco jiná jádra ještě pracují. V následujícím textu jsou často zmiňována jádra procesoru, na kterých jsou spouštěna vlákna výpočtů. V textu je pro jednoduchost uvažováno, že na jednom jádru může běžet pouze jedno vlákno, i když je v praxi často používán hyper-threading.

V průběhu návrhu a implementace bylo vytvořeno několik verzí algoritmu, které se odlišovaly tím, jak docílit, aby byla všem jádrům procesoru přidělována práce rovnoměrně. Nakonec byly vytvořeny dvě verze paralelního quicksortu. První z nich vytváří nové vlákno při každém rozdělení posloupnosti na dvě menší, dokud není rozdělovaná posloupnost příliš malá. Druhá verze paralelního quicksortu používá pouze malý počet vláken, kterým opakovaně přiděluje práci.

V první verzi je vyvažování zátěže docíleno tím, že je vytvořeno mnoho vláken s menším objemem práce. Díky vysoké granularitě jsou vytvořená vlákna rovnoměrně rozdělena mezi jednotlivá jádra procesoru. Ve druhé verzi je vyvažování zátěže docíleno tím, že vlákna sdílí frontu s úlohami. Díky tomu se nestane, že by výpočet na jednom jádru byl ukončen výrazně dříve, než výpočty na ostatních jádrech.

Tato kapitola je věnována návrhu a implementaci paralelního quicksortu, který nevyužívá nestandardních rozšíření jazyka C++ nebo externích knihoven. Na začátku této kapitoly jsou popsány postupy, které byly použity pro rovnoměrné rozdělení zátěže mezi jednotlivá jádra procesoru. Tedy jak je řešena správa vláken – jejich vytváření a přidělování jim práce. V další části

je popsán návrh celého algoritmu, který umožňuje udržitelnost a soudružnost dvou verzí paralelního quicksortu. Ve třetí části je popsán základní princip fungování navrženého algoritmu. V závěru kapitoly jsou popsány implementované kombinace s jinými řadícími algoritmy.

Podstatnou součástí implementace paralelního quicksortu je i implementace algoritmu pro rozdělování, který je popsán v kapitole 3.4.

4.1 Správa vláken

U paralelního algoritmu je nutné vyvažování zátěže tak, aby všechna jádra zpracovávala přibližně stejný objem práce. Implementace paralelního quicksortu vytváří vlákna, která operační systém přiřazuje jednotlivým jádrům procesoru. Během návrhu a implementace bylo prozkoumáno několik možností, jak řídit vytváření vláken a jak jim přidělovat práci, aby rozvržení zátěže bylo optimální.

Prvním možným způsobem rozdělení práce je rekurzivní rozdělení řazené posloupnosti na p částí, kde p je počet dostupných jader procesoru. Je tedy vytvořeno p vláken. Každé vlákno je zodpovědné za seřazení jedné podposloupnosti. Tento naivní přístup však vede k nevyvážení zátěže jednotlivých jader. Řazení podposloupností může být diametrálně odlišně náročné. A to i za předpokladu, že jsou podposloupnosti stejně velké.

Jako řešení nevyvážení zátěže se nabízí rozčlenění řazené posloupnosti na více částí, než je počet jader. Díky vyšší granularitě je možné dosáhnout lepšího vyvážení zátěže.

Po provedení experimentů na různých typech dat se ukázaly jako nejlepší dvě níže popsané verze, které i přes svoji odlišnost dosahují podobně dobrých výsledků.

Ještě před představením samotných verzí správy vláken je nutné zde uvést, které části algoritmu mohou být paralelizovány bez nutnosti synchronizace a ve kterých částech je nutné vlákna synchronizovat.

4.1.1 Synchronizace vláken

Časově významnou součástí paralelního quicksortu je řazení menších podposloupností, které vznikly rekurzivním rozdělováním řazené posloupnosti na dvě části. Při tomto řazení každé vlákno řadí sekvenčně svoji podposloupnost, jak je popsáno v částech 1.2 a 1.3. V případě tohoto paralelního zpracovávání menších podposloupností individuálními vlákny není třeba žádná synchronizace mezi vlákny. Řazení podposloupností je na sobě nezávislé.

Odlišná situace nastává u paralelního rozdělování. V průběhu paralelního rozdělování je na několika místech algoritmu nutná synchronizace vláken, jak je popsáno v části 3.4. Pro další výpočty je třeba, aby všechna vlákna dospěla do určitého bodu algoritmu rozdělování. Až poté mohou pokračovat v provádění dalších výpočtů. Proto je vhodné, aby všechna vlákna pracující na

daném rozdělování běžela současně. K synchronizaci vláken se nabízí využití tzv. bariéry.

Bariéra je druh synchronizační techniky. Představuje místo v běhu programu, kde na sebe vlákna čekají. Pokud vlákno dorazí v programu na místo, kde se bariéra nachází, musí zastavit svůj běh. Pokračovat může pouze tehdy, pokud této bariéry dosáhnou i všechna ostatní vlákna, která se daného výpočtu účastní.

V rozšíření OpenMP je možné implicitní i explicitní používání bariér. Implicitní bariéry se nachází například na konci paralelního regionu. Explicitně je možné vytvořit bariéru pomocí direktivy `#pragma omp barrier`. Tato bariéra čeká na všechna vlákna běžící paralelně v daném regionu.

Ve jazyce C++ neexistuje standardní konstrukt pro bariéru. Bariéra se v současné době nachází pouze v experimentální části.

Z toho důvodu byla naimplementována vlastní bariéra. Bariéry se používají v paralelním rozdělování. Zároveň může běžet více paralelních rozdělování různých podposloupností. Proto je bariéra sdílena mezi vlákny, která na sebe mají čekat.

Rozdělování je nutné synchronizovat na více místech algoritmu. Z toho důvodu je bariéra implementována tak, aby mohla být použita opakovaně. V implementaci bariéry jsou použity podmíněné proměnné. Vlákna tedy na sebe čekají pasivně a nespotřebovávají zbytečně procesorový čas.

4.1.2 Velké množství vláken

Quicksort je rekurzivní algoritmus, který rozděluje řazenou posloupnost na dvě podposloupnosti, které následně nezávisle rekurzivně řadí. Přirozeně se zde nabízí tzv. úlohový paralelizmus. Každá podposloupnost tvoří novou úlohu, kterou je možné nezávisle zpracovat. Jak bylo řečeno v kapitole 2.1, C++11 vláknům není možné jednoduše opětovně přiřazovat práci. Nabízí se tedy pro zpracování podposloupností vytvořit dvě nová vlákna, která podúlohy zpracují.

V rámci optimalizací je vhodné jednu z podposloupností zpracovat současným vláknem, protože to již žádnou další práci vykonávat nemusí. Ušetří se tím vytváření nového vlákna, což je časově relativně náročná operace.

Při vytváření velkého množství vláken je třeba se rozhodnout, do kdy mají být vytvářena pro podposloupnosti nová vlákna a kdy se již vyplatí řadit posloupnost sekvenčně. Dřívější přepnutí do sekvenční verze šetří náklady spojené s vytvářením nových vláken a rekurzivním volání funkcí. Naopak pozdějším přepnutím do sekvenční verze je možné dosáhnout vyšší granularity a tím pádem i lepšího rozložení zátěže mezi jednotlivá jádra procesoru. Rozhodnutí, kdy se již vyplatí podposloupnost řadit sekvenčně, je možné provádět dvěma způsoby.

Prvním možným způsobem je stanovení fixního počtu vláken, která budou za běhu programu vytvořena. Nechť je tento celkový počet vláken označen

jako p. Celá posloupnost A je řazena p vlákny. Posloupnost A je rozdělena na dvě podposloupnosti A_1 a A_2. Následně je p vláken rozděleno na dvě části p_1 a p_2 a je rekurzivně zavoláno řazení posloupnosti A_1 s p_1 vlákny a řazení posloupnosti A_2 s p_2 vlákny. Rozdělení p vláken na dvě části může být proporcionální podle poměru velikostí podposloupností A_1 a A_2. Pokud počet vláken v aktuálně řazené posloupnosti klesne na jedno vlákno, je dále tato posloupnost řazena sekvenčně.

Druhým možným způsobem je stanovení prahu pro sekvenční řazení. Pokud velikost řazené posloupnosti klesne pod tuto mez, je tato posloupnost dále řazena sekvenčně. Toto řešení může vést k vytvoření vyššího počtu vláken. Na druhou stranu zabrání tomu, aby některé vlákno řadilo sekvenčně příliš dlouhou podposloupnost.

Praktickými experimenty se druhý uvedený způsob ukázal jako vhodnější. Proto je ve výsledné implementaci přecházeno k sekvenčnímu řazení, pokud velikost řazené posloupnosti klesne pod určitou mez. Tato mez může být specifikována při spuštění řazení. Bylo experimentováno s různými metodami, jak stanovit výchozí nastavení meze. Jednou z vyzkoušených variant bylo například její stanovení na základě velikosti řazené posloupnosti a počtu jader tak, aby na každé jádro procesoru připadl v průměrném případě určitý počet úloh. Tato závislost byla dána vztahem $n/(p \cdot t)$, kde n je velikost řazené posloupnosti, p je počet jader procesoru a t je počet úloh připadajících na jedno jádro. Při experimentálním měření se však jako nejlepší pro různé druhy vstupů ukázalo stanovení meze na konstantní hodnotu, konkrétně na padesát tisíc prvků.

Za běhu programu je vytvořeno výrazně větší množství vláken, než kolik je jader procesoru. Jednotlivá vlákna mají relativně malé množství práce, kterou musí vykonat. Díky tomu je postupným spouštěním vláken práce rovnoměrně rozdělována mezi jádra procesoru.

4.1.3 Fond vláken

Vytváření velkého počtu vláken je výpočetně náročné. Režie vytváření nového vlákna je přibližně 2000krát pomalejší než zavolání funkce [25]. Standard C++ nespecifikuje, jak mají být vlákna vytvářena. Je pouze specifikováno, že konstruktor `std::thread` by se měl chovat tak, jako by bylo vytvořeno nové vlákno operačního systému. Není tedy jisté, zda systém vytváří pokaždé zcela nové vlákno, či je určitým způsobem schopný recyklovat vlákna stará. Obvykle je konstrukt `std::thread` implementován pouze jako minimální nadstavba nad systémovým vytvářením vlákna.

C++ v současné verzi neobsahuje standardní implementaci fondu vláken, který by umožnil znovupoužití vytvořených vláken. Z nestandardních knihoven se fond vláken vyskytuje například v knihovně boost, kde dovoluje opětovné přidělování práce vytvořeným vláknům. Ve standardu však podobný konstrukt chybí.

Z důvodu absence fondu vláken ve standardní knihovně byl implementován vlastní fond vláken. Ten umožňuje vytvořeným vláknům opakovaně přidělovat práci. Díky tomu je v implementaci paralelního quicksortu možné vytvořit pouze malý počet vláken, odpovídající počtu jader procesoru. Vyvažování zátěže je pak docíleno dynamickým přidělováním menších kousků práce jednotlivým běžícím vláknům.

Základem fondu vláken je synchronizovaná fronta. Do fronty jsou vkládány úlohy určené ke zpracování. Jednotlivá vlákna si úlohy z fronty vybírají. Vhodným použitím mutexů a atomických proměnných je garantované, že vkládání i vybírání úloh je bezpečné i ve více-vláknovém prostředí. Tedy že se například nestane, že by dvě vlákna získala tu samou úlohu.

Samotný fond vláken obsahuje pole vláken. Po vytvoření vlákna čekají na signál, že mohou začít zpracovávat frontu úloh. Každé vlákno si vybere jednu úlohu z fronty, pokud je nějaká úloha k dispozici. Po dokončení zpracování úlohy se vlákno pokusí získat další úlohu z fronty. Pokud se ve frontě žádná další úloha nenachází, vlákno se uspí. I když je fronta prázdná, není možné ještě ukončit činnost vlákna. Je totiž možné, že v budoucnu mohou být vygenerovány nové úlohy vlákny, které ještě zpracovávají své úlohy. Pokud je nějaká úloha vložena do fronty, je probuzeno jedno spící vlákno, které danou úlohu zpracuje. Když všechna vlákna dokončí svoji práci, dostanou všechna vlákna signál, že bylo zpracovávání všech úloh dokončeno. V tu chvíli všechna vlákna ukončí svoji činnost.

V průběhu algoritmu jsou vlákna využívána ve dvou kontextech. Prvním je samotné řazení rozdělených podposloupností. Úlohy vytvořené v tomto kontextu nevyžadují žádnou synchronizaci. Pokud všechna vlákna dokončila své úlohy, mohou všechna vlákna ukončit svoji činnost.

Složitější je situace u vytvořených úloh v kontextu druhém. Jedná se o úlohy, které jsou vytvořeny v rámci paralelního rozdělování posloupnosti. Tyto úlohy musí běžet současně, protože je během rozdělování nutná jejich synchronizace na několika místech. Pokud by tyto úlohy byly ukládány do stejné fronty, jako jsou ukládány úlohy z prvního kontextu, mohlo by dojít k uváznutí spuštěných úloh. Uváznutí by mohlo nastat, pokud by úlohy ze druhého kontextu byly proloženy úlohami z prvního kontextu a nebylo by tak možné najednou spustit všechny úlohy nutné pro rozdělování, nebo by některá vlákna musela čekat dlouhou dobu na dokončení vložené úlohy z prvního kontextu.

Z toho důvodu jsou v implementaci použity dva fondy vláken. Jeden slouží ke zpracovávání úloh z prvního kontextu, druhý slouží ke zpracovávání úloh ze druhého kontextu.

Při zpracovávání úloh z druhého kontextu je nutné brát v úvahu ještě jednu odlišnou vlastnost. V případě, že všechna vlákna dokončí provádění svých úloh, nesmí se vlákna, na rozdíl od prvního kontextu, ukončit. Po ukončení rozdělování jedné podposloupnosti je pravděpodobné, že některá další podposloupnost bude potřebovat provést paralelní rozdělování. Avšak úlohy pro rozdělování nemusí být ještě vygenerovány v době ukončení rozdělování před-

chozí podposloupnosti. Proto jsou vlákna, která jsou využívána ke zpracování úloh ve druhém kontextu, uspána i poté, co už žádné z nich nepracuje. Pokud je za běhu algoritmu zjištěno, že již nebudou vznikat žádné další úlohy z prvního kontextu, je jisté, že už nebudou vznikat ani úlohy z kontextu druhého. V tom okamžiku jsou i vlákna zpracovávající úlohy ze druhého kontextu notifikována, že mohou ukončit svoji činnost.

Komunikace mezi jednotlivými vlákny fondu vláken je realizována pomocí mutexů a podmíněných proměnných z knihovny thread, které jsou popsány v části 2.1. Díky těmto synchronizačním technikám je možné jednotlivá vlákna informovat o vložení nové úlohy do fronty nebo o tom, že mohou ukončit svoji činnost. Obdobně tato vlákna mohou podat informaci o dokončení zpracovávání všech dostupných úloh.

4.2 Návrh programu

Implementace paralelního algoritmu quicksort je vytvořena ve dvou verzích, které různě nakládají s vytvářením vláken. První z nich vytváří mnoho vláken, druhá jich vytváří méně, protože používá fond vláken. Obě verze mají mnoho společného kódu, přesto se ve většině funkcí vyskytují zásadní odlišnosti. Například při rekurzivním volání zpracovávání podposloupností první verze vytváří jedno nové vlákno. Druhá verze naopak vkládá úlohu do fronty. Z důvodu udržitelnosti kódu je třeba, aby co největší část kódu byla mezi oběma verzemi sdílena.

Možností, jak zabránit duplicitnímu kódu ve dvou verzích algoritmu, je několik. Jednou z možností je užití polymorfizmu. Abstraktní třída pro řazení by mohla být specializována jednou pro verzi s velkým počtem vláken a jednou pro verzi s fondem vláken. Řešení s polymorfizmem by bylo vhodné z hlediska správného návrhu programu, ale nebylo by příliš efektivní, ani konzistentní s podobou standardních knihoven.

Další možností je použití podmíněného překladu. Při překladu by bylo specifikováno, která varianta správy vláken má být použita. Toto řešení by bylo optimální z hlediska efektivity běhu programu. Samotný kód by byl však při tomto přístupu hůře čitelný.

Jinou možností je vyextrahování společné funkcionality obou verzí do samostatných funkcí. Tím by byly duplicity zdrojových kódů obou verzí z velké části odstraněny. Vzhledem k odlišnostem v mnoha funkcích by ale i tak bylo třeba vytvořit dvě prakticky totožné kostry programu.

Poslední možností je vytvoření společného kódu pro řazení, který je parametrizován třídou. Jedná se prakticky o opačný přístup než v možnosti předchozí. Společná funkcionalita tvoří základ programu a odlišné části se řeší ve specializovaných metodách tříd.

Z různých vyzkoušených druhů návrhu byla nakonec zvolena poslední možnost, kde je společný kód pro řazení parametrizován třídou. Tato možnost

byla vybrána, protože se jedná o relativně dobře čitelné řešení, ve kterém jsou duplicity kódu minimalizovány.

V implementaci byly vytvořeny třídy pro správu vláken. Jedna z nich obsahuje kód specifický pro správu vláken bez fondu vláken. Druhá třída obsahuje kód pro správu vláken s fondem vláken. Třídy mají stejné hlavičky metod, které jsou volány ze společné části programu. Objekt třídy pro správu vláken je předáván parametrem do společné části řadícího programu. Typ použité třídy pro správu vláken je určen šablonovým parametrem.

Odlišnost implementace obou verzí je patrná hlavně na třech místech. Prvním je prvotní spuštění rekurzivního řazení. Ve verzi bez fondu vláken je pouze zavoláno rekurzivní řazení posloupnosti. Ve verzi s fondem vláken je tato úloha vložena do fronty a vlákna jsou notifikována, že mohou začít se zpracováváním úloh. Druhým odlišným místem je rekurzivní volání řazení podposloupností. V jedné verzi je vytvářeno vlákno, ve druhé je vložena úloha do pracovní fronty. Posledním rozdílným místem je bod, ve kterém je spouštěno paralelní rozdělování posloupnosti. Ve verzi bez fondu vláken jsou vytvořena nová vlákna pro rozdělování a následně se čeká na jejich připojení. Ve verzi s fondem vláken je používán již existující fond vláken pro paralelní rozdělování. Z důvodu použití jednoho fondu vláken s frontou úloh pro rozdělování je nutné, aby zároveň neprobíhalo více rozdělování najednou. Na dokončení rozdělování vlákny se čeká s využitím podmíněné proměnné.

4.3 Základní princip navrženého algoritmu

Základem navržené implementace paralelního algoritmu quicksort je rekurzivní funkce `ParallelSortQsConquer`, jejíž kostra je znázorněna v algoritmu 4.1. Tato funkce zajišťuje rozdělení řazené posloupnosti a následné rekurzivní zavolání sebe sama na dvě vzniklé podposloupnosti. V ukázce implementace je viditelné i použití šablonového parametru `ThreadManager` pro určení typu správy vláken, který je zmíněn v textu výše.

Ve funkci `ParallelSortQsConquer` je nejprve zjištěno, jestli je řazená posloupnost dostatečně malá na to, aby byla řazena sekvenčně jen jedním vláknem. Posloupnost je považována za dostatečně malou, pokud je její délka menší než určitá mez. Tato mez je ve výchozím nastavení stanovena na padesát tisíc prvků. Hodnota této meze byla stanovena na základě provedených experimentů, jejichž výsledky jsou popsány v části 5.3.3.

Pokud není řazená posloupnost dostatečně malá, jsou její prvky přeuspořádány na dvě části. V první části se nachází prvky menší než pivot a ve druhé prvky větší než pivot. Pivot je stanoven jako medián třiceti prvků vybraných z posloupnosti. Počet prvků, ze kterých je vybírán medián, byl stanoven na základě provedených experimentů. Samotné rozdělování probíhá paralelně, pokud je k dispozici více vláken. Pokud je k dispozici pouze jedno vlákno, rozdělování probíhá sekvenčně. Rozdělování posloupnosti na dvě probíhá i po-

kud je původní posloupnost rozdělena na více částí, než je počet vláken. Díky tomu vznikne více úloh, které mohou vlákna zpracovávat. Dochází tak k lepšímu vyvažování zátěže. Princip rozdělování je popsán v části 3.4.

Po rozdělení posloupnosti tedy vzniknou dvě nové úlohy, které mohou vlákna dále zpracovávat. Koncové volání funkce `RunConquer` obaluje rekurzivní zavolání řazení na obě podposloupnosti, tedy zpracování dvou vzniklých úloh. Konkrétní implementace této funkce je odlišná pro různé typy správy vláken. Ve vytvořené implementaci není použito omezení hloubky rekurze, které je popsáno v části 1.2.3. Toto omezení není použito, protože vzhledem k výběru pivota jako mediánu ze třiceti vzorků je riziko opakovaně výrazně nerovnoměrného rozdělení posloupnosti minimální.

```
template<typename RAIter,typename Compare, typename ThreadManager>
void ParallelSortQsConquer(RAIter first, RAIter last,Compare comp,
     std::size_t num_threads,std::size_t sequential_threshold,
    ThreadManager &thread_manager) {
     (...)
     DifferenceType n = last - first;
     if (n <= sequential_threshold) {
          std::sort(first, last, comp);
          return;
     }

     DifferenceType split = ParallelSortQsDivide(...);
     std::size_t num_threads_left = std::max((num_threads * split)
         / n, 1lu);

     thread_manager.RunConquer(first, last, comp, split,
         num_threads, num_threads_left, sequential_threshold,
         thread_manager);
}
```

Algoritmus 4.1: Základní rekurzivní funkce implementace paralelního algoritmu quicksort.

4.4 Kombinace s jinými řadícími algoritmy

Během návrhu a implementace byly prozkoumány možnosti kombinace paralelního quicksortu s jinými řadícími algoritmy. Jak již bylo zmíněno dříve, paralelní quicksort je za určitých podmínek přepnut na sekvenční quicksort. Právě sekvenční quicksort se obvykle kombinuje s jinými algoritmy. Kombinacemi se zabraňuje přílišnému zanoření a rekurzivnímu řazení příliš malých posloupností.

Níže popsané kombinace s jinými algoritmy byly implementovány. Během měření se následně ukázalo, že implementace sekvenčního quicksortu ve stan-

dardní knihovně je natolik optimální, že se odlišnými implementacemi nepodařilo dosáhnout lepších výsledků, než za použití standardního řadícího algoritmu. Proto je ve finální implementaci použit právě standardní knihovní sekvenční `std::sort`. Ten také implementuje kombinaci s algoritmy heapsort a insert sort.

4.4.1 Heapsort

Kombinace s algoritmem heapsort se používá, když hrozí, že by doba běhu programu byla příliš dlouhá. Složitost quicksortu je v nejhorším případě až kvadratická. Příčinou špatné složitosti je nerovnoměrné rozdělování posloupnosti na dvě podposloupnosti. Pokud tedy dosáhne rekurze quicksortu přílišného zanoření, přepne se algoritmus na heapsort, který má garantovanou složitost $\mathcal{O}(n \cdot \log(n))$. Maximální povolená hloubka zanoření quicksortu byla stanovena na dvojnásobek logaritmu délky posloupnosti. Je to stejná mez, která je použita v sekvenčním quicksortu ze standardní knihovny.

Kontrolovat hloubku zanoření je možné i v rekurzivní funkci paralelní části implementace quicksortu. Při zkoumání jiných implementací paralelního quicksortu bylo zjištěno, že většina implementací hloubku zanoření nekontroluje. Kontrolují ji převážně ty implementace, u kterých je pravděpodobné, že rozdělení řazené posloupnosti na dvě části nebude moc rovnoměrné. Pokud je kontrola hloubky zanoření zapnuta, zpomaluje to implementaci popisovanou v této práci o jedno až dvě procenta.

Ve výsledné implementaci, která je představena v této práci, nakonec není kontrola hloubky rekurze v paralelní části algoritmu prováděna. Není to potřeba z toho důvodu, že pivot je vybírán z velkého množství vzorků posloupnosti (při výchozím nastavení ze třiceti prvků). Díky tomu je velmi nepravděpodobné, že by řazená posloupnost byla opakovaně rozdělena výrazně nerovnoměrně.

4.4.2 Insert sort

Kombinace s algoritmem insert sort se používá pro koncové řazení malých podposloupností. Tato kombinace v paralelní části algoritmu není nutná. Malé posloupnosti nejsou paralelně řazeny, protože pokud velikost řazené posloupnosti klesne pod určitou mez, je již řazena sekvenčně. Mez pro přepnutí na nerekurzivní algoritmus je obvykle řádově kolem 16 prvků. Mez pro přepnutí z paralelní do sekvenční verze je ve výchozím nastavení experimentálně stanovena na padesát tisíc prvků.

Experimentálně bylo zjištěno, že na výkon paralelního quicksortu nemá měřitelný vliv, jestli jsou jednotlivé malé podposloupnosti řazeny insert sortem postupně, nebo až najednou nakonec. Jak je zmíněno v kapitole 1.2.4, výhodou prvního přístupu je paměťová lokalita, výhodou druhého je, že je koncový insert sort zavolán jen jednou. Z experimentálního měření plyne, že klady

a zápory obou způsobů jsou v praxi vyvážené nebo jsou jejich rozdíly prakticky zanedbatelné. Ve finální implementaci je použit standardní řadící algoritmus, který provádí jeden insert sort přes celé pole.

Kapitola 5

Testování

V průběhu vývoje implementace paralelní verze algoritmu quicksort založené na C++11 vláknech bylo důležité provést řadu experimentů a měření. Výsledky experimentů přispěly ke vhodnému výchozímu nastavení různých konstant, které jsou v implementaci použity. Implementace, která je popsána v této práci, je označována jako *C++11Sort*.

Po dokončení implementace byla podrobněji otestována efektivita implementace pro různé typy řazených dat. Zkoumán byl i vliv výchozího seřazení testovaných dat na dobu nutnou k jejich seřazení. Prozkoumán byl i vliv počtu použitých jader procesoru.

Nedílnou součástí bylo i experimentální porovnání vytvořené implementace s existujícími implementacemi paralelních in-place řadících algoritmů dostupných pro C++.

V této kapitole je představeno testovací prostředí. Následně je popsán vliv nastavení různých parametrů na rychlost řazení. V další části je porovnána efektivita implementace pro různá vstupní data. V poslední části této kapitoly je porovnána implementace, která je popsána v této práci, s ostatními existujícími implementacemi in-place řadících algoritmů.

Konkrétní hodnoty naměřených výsledků jsou uloženy na přiloženém CD.

5.1 Testovací prostředí

Měření byla prováděna na výpočetním klastru STAR. Výpočetní klastr je tvořen front-end uzlem a osmi výpočetními uzly.

Přes front-endový uzel je možné vkládat požadavky do výpočetní fronty. Fronta postupně zpracovává příchozí požadavky. Díky tomu je zajištěno, že jednotlivé procesy, které jsou určeny ke zpracování, nejsou ovlivněny jinými běžícími procesy. Proto je tento server vhodný pro měření.

Výpočetní uzly mají k dispozici 64 GB paměti a 64bitový dvacetijádrový procesor Intel Xeon E5-2630 v4. Procesor je taktován na 2.2 GHz. Mezipaměti L1, L2 a L3 mají velikost 32 KB, 256 KB, respektive 25600 KB.

Přestože uvedený procesor podporuje hyper-threading, na výpočetním serveru STAR není povolen. Maximální počet běžících vláken v jednom okamžiku je tedy dvacet. Při experimentech bylo využíváno všech dvaceti jader, pokud není uvedeno jinak.

Na serveru je operační systém CentOS 7.4.1708 64bit. Všechny programy byly kompilovány překladačem GCC ve verzi 8.2.1. Použitý standard C++ při kompilaci byl C++17.

5.2 Testovací data

Testování bylo prováděno na dostatečně velkých datech, která se však zároveň vešla do paměti na testovacím serveru STAR. Nejčastěji bylo při měření zpracováváno přibližně 16 GB dat, při některých měřeních bylo zpracováváno i 32 GB dat. Velké objemy vstupních dat byly voleny záměrně, aby výpočet trval dostatečně dlouhou dobu a bylo tak možné využít smysluplně všechna jádra procesoru.

5.2.1 Generování dat

Vzhledem k objemu zpracovávaných dat jsou vstupní data generována za běhu programu. Generování dat za běhu programu je výrazně rychlejší než jejich načítání ze souboru. Například načítání dvou miliard 64bitových čísel ze souboru trvalo 245 sekund, zatímco jejich generování pouze 20 sekund.

Data jsou generována tzv. *Mersenne Twister 19937* pseudonáhodným generátorem čísel ze standardní knihovny <`random`>. Vzhledem k velikosti generovaných dat je generování prováděno paralelně všemi dostupnými vlákny. Kvůli opakovatelnosti měření je v implementaci použita fixní inicializační hodnota generátoru (seed). Každé vlákno musí mít svoji vlastní inicializační hodnotu. Pokud by tomu tak nebylo, vlákna by generovala stejná data.

5.2.2 Použitá data

Testování probíhalo na různých datech, která se lišila svým typem, výchozím uspořádáním nebo počtem nabývaných hodnot. Mezi významné typy dat, nad kterými bylo provedeno měření, patří:

- celá čísla (int),
- desetinná čísla s dvojitou přesností (double),
- znakové řetězce (string).

Počet řazených dat byl stanoven tak, aby se data vešla do paměti. V případě celých čísel byla nejčastěji měření prováděna na posloupnostech, které obsahovaly dvě miliardy čísel. V případě znakových řetězců bylo obvykle řazeno 300 milionů řetězců dlouhých pět až třicet znaků.

Z hlediska uspořádání vstupních dat byla uvažována hlavně:

- náhodně řazená data,
- již seřazená data,
- opačně seřazená data.

Dále byla experimentálně vyhodnocována data, která nabývala pouze omezeného počtu hodnot. Například u celých čísel byla obvykle uvažována čísla rovnoměrně distribuována z rozsahu 0 až $2^{32} - 1$. Při testování s omezeným počtem hodnot byl počet nabývaných hodnot pouze sto.

5.3 Nastavení parametrů implementace

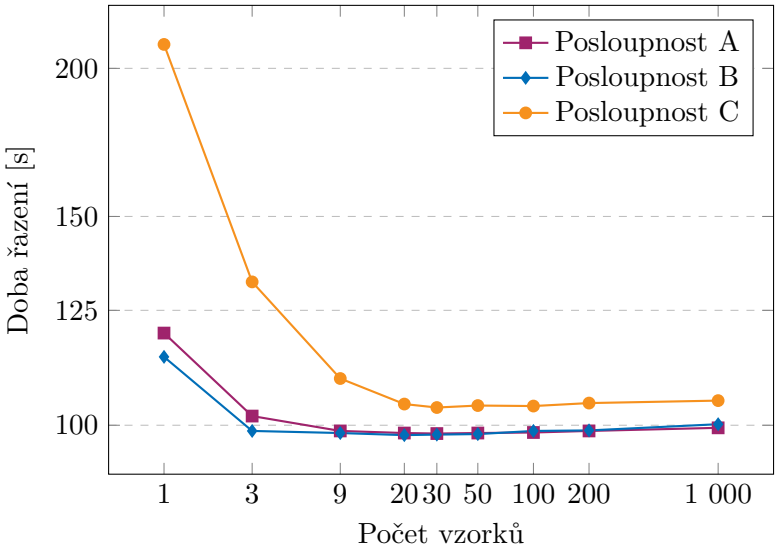
Implementace paralelního quicksortu založená na C++11 vláknech obsahuje mnoho parametrů, které je potřeba nastavit tak, aby bylo řazení co nejefektivnější. Konkrétní hodnoty parametrů byly kalibrovány experimentálně v průběhu implementace. Ve výsledné implementaci jsou použity takové výchozí hodnoty parametrů, které se jevily jako nejlepší při řazení různých typů dat. Hodnoty parametrů jsou založeny na experimentech nejen nad různými typy dat, ale i nad různými konkrétními hodnotami, aby nebyla výsledná implementace příliš přeučená (overfitted) na konkrétní řazenou posloupnost.

V následujících odstavcích jsou popsány zajímavé vlivy konkrétních parametrů na výkonnost implementace. Pokud není zmíněno jinak, všechny ostatní parametry mají hodnoty stejné jako ve výsledné implementaci. Výsledky měření v této části jsou uvedeny pro verzi bez fondu vláken, pokud není uvedeno jinak. Experimenty prováděné s verzí s fondem vláken dosahovaly pro tyto měřené veličiny podobných výsledků, proto zde nejsou uvedeny.

5.3.1 Výběr pivota

Při rozdělování řazené posloupnosti na dvě podposloupnosti je třeba určit pivota. Ten je nejčastěji vybírán jako medián několika vybraných hodnot z řazené posloupnosti. Při výběru pivota pouze z jednoho prvku řazené posloupnosti je vysoká pravděpodobnost nerovnoměrného rozdělení posloupnosti, a tedy v důsledku i neefektivního řazení. Pokud je pivot vybírán jako medián z mnoha vzorků posloupnosti, je pravděpodobnost rovnoměrného rozdělení posloupnosti vysoká. Na druhou stranu hledání mediánu mnoha vzorků spotřebovává výpočetní čas. V praxi je často pivot určen jako medián ze tří mediánů tří vybraných prvků.

Experimentálně byla ověřena výkonnost implementace při výběru pivota z různého počtu vzorků. Vliv počtu vzorků na rychlost řazení dvou miliard celých čísel je zobrazen na obrázku 5.1. Na obrázku jsou znázorněny naměřené



Obrázek 5.1: Vliv počtu vzorků, ze kterých je vybírán pivot, na dobu řazení tří různých posloupností celých čísel. Posloupnosti obsahují dvě miliardy prvků.

hodnoty pro tři různé posloupnosti, protože je tento experiment velmi závislý na hodnotách řazených dat.

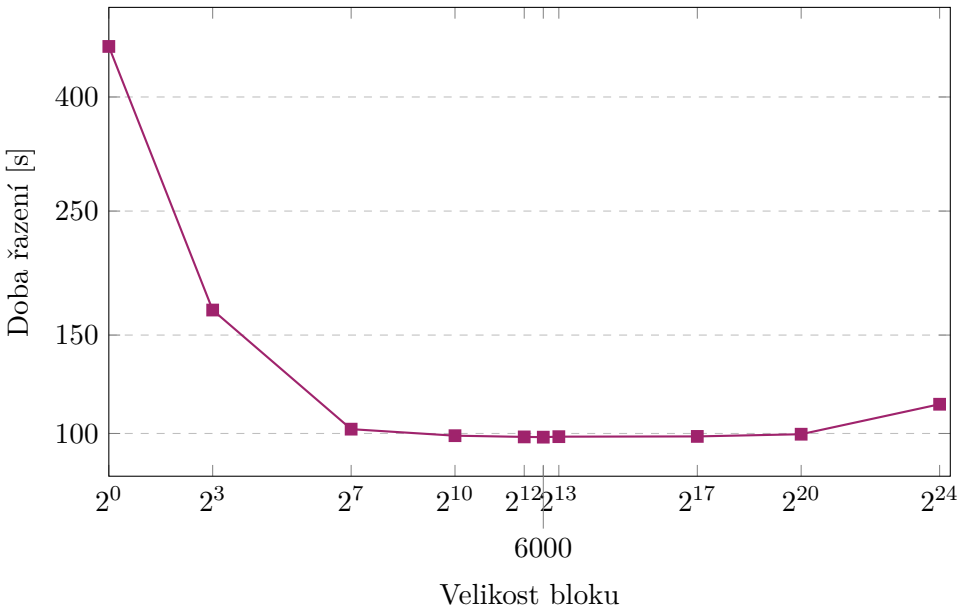
Pokud byl pivot vybírán pouze z jednoho vzorku, byla implementace v průměrném případě více než dvojnásobně pomalejší oproti výsledné implementaci. Výběr pivota ze tří prvků značně snížil dobu běhu algoritmu. Nejlepších výsledků bylo dosaženo, pokud byl pivot určen jako medián z přibližně třiceti vzorků. Větší počet vzorků již výpočet neurychloval. Naopak výpočet byl s nadále se zvyšujícím počtem vzorků pomalejší, kvůli vyšší náročnosti nalezení mediánu z většího množství prvků.

Z grafu je také patrné, že pro malé počty vzorků je algoritmus více citlivý na vstupní data. Při výběru pivota pouze z jednoho vzorku byla doba řazení posloupnosti „C“ téměř dvojnásobná oproti době řazení posloupnosti „B“. Při použití výběru z více vzorků byly časové rozdíly mezi řazením jednotlivých posloupností výrazně menší.

5.3.2 Velikost bloku

V průběhu paralelního rozdělování posloupnosti na dvě podposloupnosti získávají jednotlivá vlákna prvky z pravé a prvky z levé části posloupnosti. Prvky jsou získávány ne po jednom, ale po blocích. Díky získávání prvků po blocích nemusí vlákna tak často čekat kvůli nedostupnosti prostředků.

Při používáních menších bloků je nutná častější synchronizace mezi vlákny. Na druhou stranu jsou menší bloky lepší pro vyvažování zátěže mezi jednotlivá jádra procesoru.



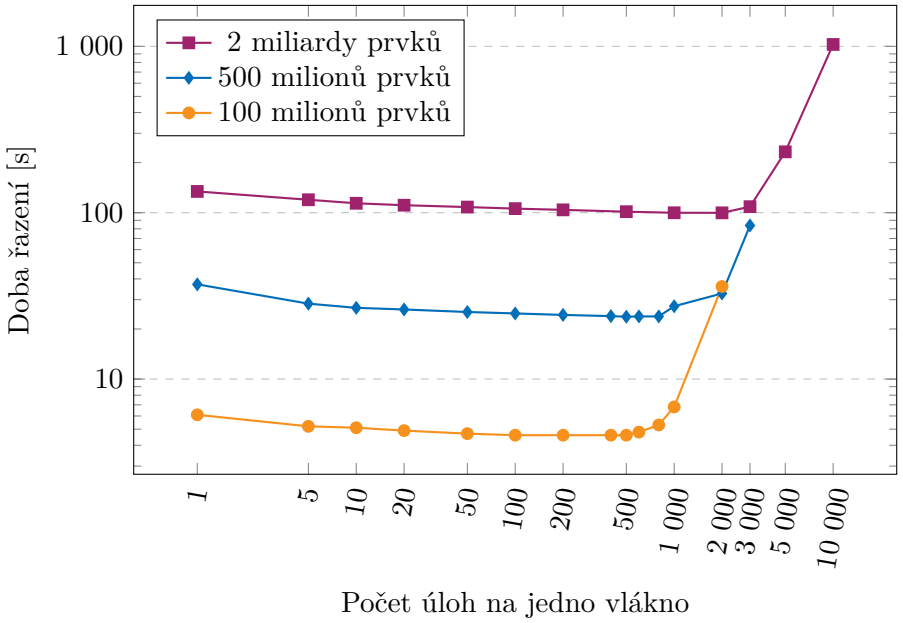
Obrázek 5.2: Vliv velikosti bloků užívaných v paralelním rozdělování na dobu řazení dvou miliard čísel.

Byla provedena experimentální měření, která si kladla za cíl zjistit nejvhodnější velikost bloku. Z výsledků experimentů se jako nejvhodnější jeví používání bloků o velikosti 6000 prvků. Při nastavení velikosti bloků na velikosti používané v jiných implementacích, například na 1024 prvků, dosahuje program podobně dobrých výsledků. Bloky výrazně menší, nebo naopak výrazně větší, zpomalují běh programu dle očekávání. Závislost doby běhu algoritmu na velikosti bloků při řazení dvou miliard čísel je vyobrazena na obrázku 5.2.

5.3.3 Práh přepnutí na sekvenční řazení

Podstatou algoritmu je rozdělení řazené posloupnosti na menší podposloupnosti (úlohy), které jsou dále zpracovávány jednotlivými vlákny sekvenčně. Důležité je najít správnou velikost těchto sekvenčně řazených úloh. Pokud by úlohy byly příliš velké, nemohla by být práce rovnoměrně distribuována mezi jednotlivá jádra procesoru. Pokud by podposloupnosti byly naopak hodně malé, docházelo by zbytečně k režii s vytvářením mnoha úloh a s jejich přiřazováním výpočetním vláknům.

Bylo vyzkoušeno několik metod, jak stanovovat výchozí nastavení prahu pro přepnutí na sekvenční řazení. Nejprve se zdálo vhodné vypočítávat hodnotu prahu v závislosti na velikosti řazené posloupnosti a počtu jader tak, aby na každé jádro připadal alespoň určitý počet úloh. Vzorec pro výpočet tohoto prahu je $n/(p \cdot t)$, kde n je velikost řazené posloupnosti, p je počet



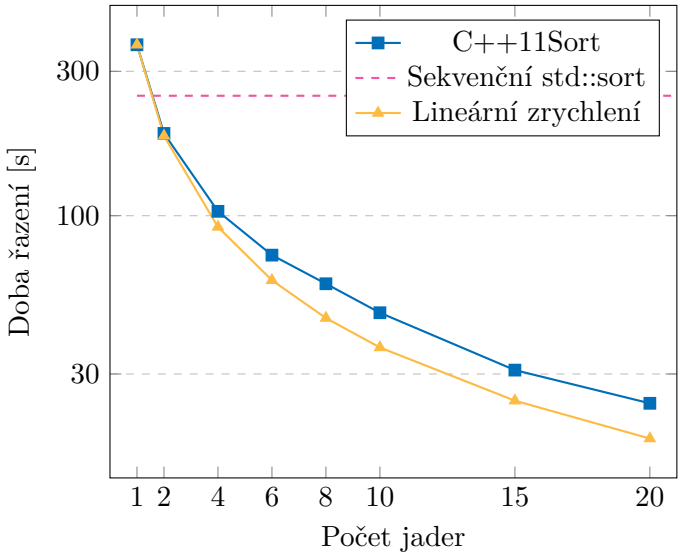
Obrázek 5.3: Vliv počtu úloh připadajících na jedno vlákno na dobu řazení různě velký posloupností.

dostupných jader a t je počet úloh na jádro procesoru. Pokud velikost posloupnosti klesne pod tento práh, není tato posloupnost dále rozdělována na menší, ale je sekvenčně seřazena jedním vláknem. V průběhu experimentů se však ukázal jako nejvhodnější konstantně stanovený práh pro přepnutí na sekvenční řazení. Z experimentů vyplynulo, že nejvhodnější hodnotou pro všeobecné použití je padesát tisíc prvků.

Na obrázku 5.3 je znázorněn vliv počtu úloh připadajících na jedno vlákno na dobu běhu programu pro různě velké řazené posloupnosti. Z výsledků měření vyplynulo, že stanovení prahu dle počtu úloh připadajících na jedno vlákno není nejvhodnějším postupem. Naopak pro všechny velikosti dosahovalo řazení nejlepších výsledků, pokud byl práh stanoven na přibližně padesát tisíc prvků. Podobných výsledků bylo dosaženo i při měřeních na osobním počítači s osmi-jádrovým procesorem. Proto byla výchozí hodnota prahu pro přepnutí na sekvenční řazení stanovena na padesát tisíc prvků.

5.3.4 Počet použitých jader

Experimentální měření probíhala na výpočetním serveru STAR, přičemž maximální počet používaných jader procesoru byl dvacet. Na serveru STAR byl vypnut hyper-threading, tedy na jednom jádru mohlo současně běžet pouze jedno vlákno. Dalším experimentem bylo zjišťováno, jaký vliv má počet použitých jader na dobu běhu programu.

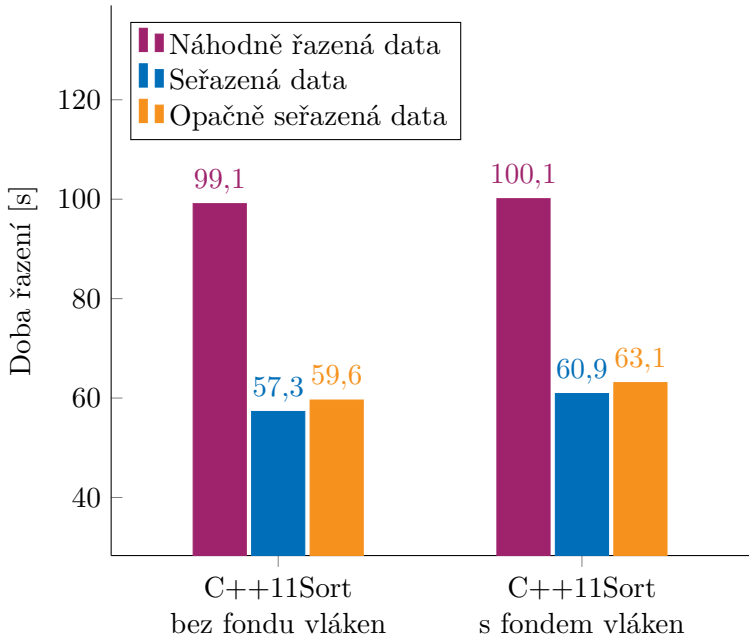


Obrázek 5.4: Vliv počtu použitých jader na dobu řazení 500 milionů čísel. V grafu je zobrazeno i teoretické lineární zrychlení algoritmu.

Měření bylo provedeno na 500 milionech celých čísel. Výsledky měření závislosti doby běhu programu na počtu použitých jader jsou zobrazeny na obrázku 5.4. Pro srovnání je v grafu uvedena doba běhu algoritmu C++11Sort bez fondu vláken a doba běhu standardního sekvenčního řadícího algoritmu `std::sort`. Doba běhu algoritmu C++11Sort s fondem vláken byla pro tato měření srovnatelná s dobou běhu verze bez fondu vláken. Verze s fondem vláken byla pro všechny počty vláken nepatrně pomalejší. Zpomalení však bylo maximálně o 1,3 %, proto je pro ilustrační účely dostačující vyobrazení pouze verze bez fondu vláken.

Z grafu je patrné, že při použití pouze jednoho jádra byl vytvořený paralelní C++11Sort 1,7krát pomalejší než sekvenční `std::sort`. C++11Sort dosahoval lepších výsledků než sekvenční `std::sort`, pokud byla použita alespoň dvě jádra. Při použití čtyř jader bylo paralelní řazení přibližně dvojnásobně rychlejší než řazení standardním sekvenčním algoritmem. Při použití dvaceti jader bylo paralelní řazení rychlejší téměř devítinásobně.

Z grafu je možné vyčíst, že je C++11Sort relativně dobře škálovatelný. Při řazení 500 milionů prvků na dvaceti jádrech dosáhl šestnáctinásobného zrychlení oproti spouštění pouze na jednom jádru. Pro lepší představu jsou v grafu uvedeny časy, kterých by paralelní algoritmus dosahoval, pokud by dosahoval lineárního zrychlení. Doba řazení 500 milionů prvků na dvaceti jádrech je relativně krátká. Například při řazení dvou miliard prvků jsou jádra využita rovnoměrněji a zrychlení C++11Sortu na dvaceti jádrech je přibližně devatenáctinásobné.



Obrázek 5.5: Vliv uspořádání dat na dobu řazení dvou miliard celých čísel.

5.4 Otestování vytvořené implementace

Cílem této práce je, aby vytvořený algoritmus C++11Sort byl všeobecně využitelný pro různé druhy řazených dat. Proto testovací měření probíhala na různých typech vstupních posloupností. Posloupnosti se lišily datovými typy svých prvků, výchozím seřazením nebo mohutností oboru hodnot.

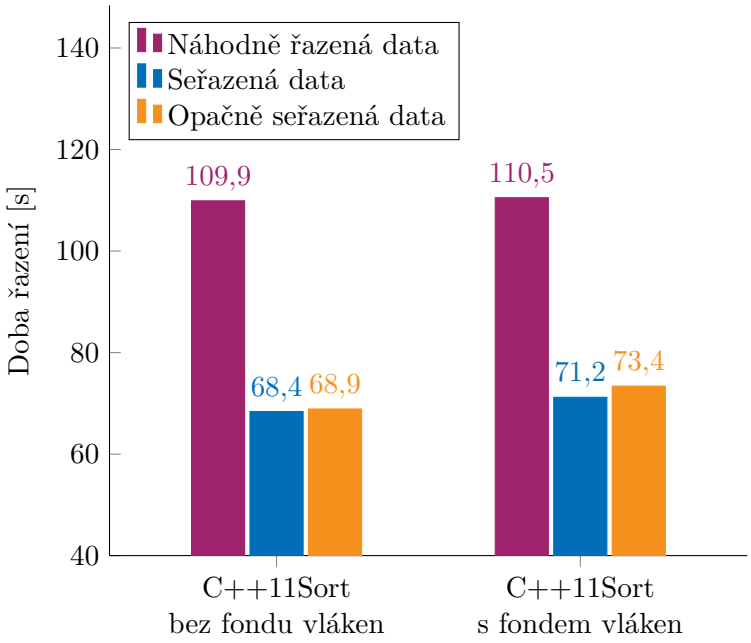
5.4.1 Vliv uspořádání vstupních dat

Dobu běhu paralelních řadících algoritmů často ovlivňuje počáteční seřazení vstupní posloupnosti. Mezi nejzajímavější vstupní uspořádaní byla vybrána náhodně seřazená data, data která jsou již seřazena a data, která jsou seřazena obráceně, než jak mají být seřazena na výstupu.

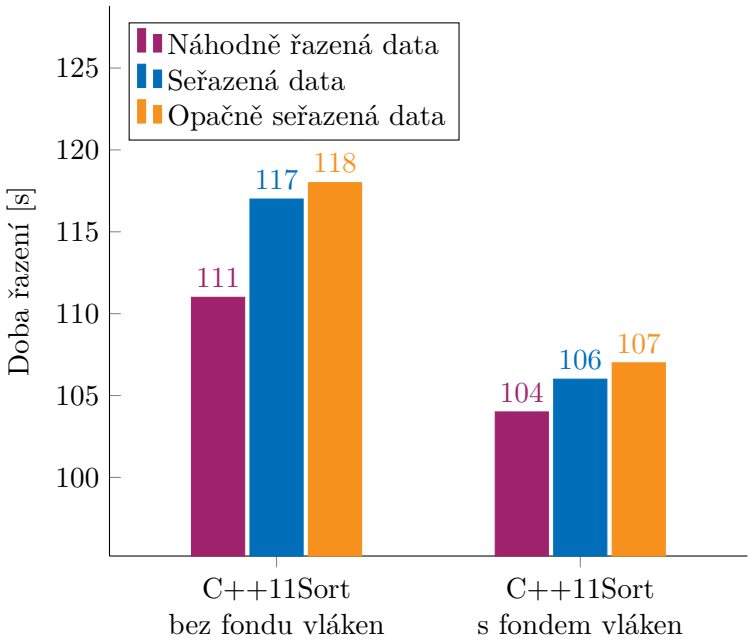
Měření proběhlo nad třemi posloupnostmi, jejichž prvky jsou různých datových typů – celá čísla, desetinná čísla a znakové řetězce. Pro každý datový typ byla provedena měření na třech výše zmíněných typech seřazení vstupních dat. Na přiložených grafech jsou znázorněny výsledky měření pro verzi C++11Sortu s fondem vláken i bez fondu vláken.

Výsledky měření při řazení dvou miliard celých čísel jsou znázorněny na obrázku 5.5. Vliv vstupního seřazení na dobu řazení dvou miliard desetinných čísel je vyobrazen na obrázku 5.6. Výsledky pro řazení znakových řetězců jsou znázorněny na obrázku 5.7.

Z výsledků je patrné, že řazené desetinných čísel je trochu pomalejší než



Obrázek 5.6: Vliv uspořádání dat na dobu řazení dvou miliard desetinných čísel.



Obrázek 5.7: Vliv uspořádání dat na dobu řazení 300 milionů znakových řetězců.

řazení celých čísel. Celá i desetinná čísla se pro různá vstupní seřazení vstupních dat chovají podobně. Nejvíce časové náročné je řazení náhodné posloupnosti. Řazení již seřazených dat je téměř dvojnásobně rychlé oproti již seřazené posloupnosti. V průběhu řazení již seřazené posloupnosti není nutné v průběhu rozdělování prohazovat žádné prvky.

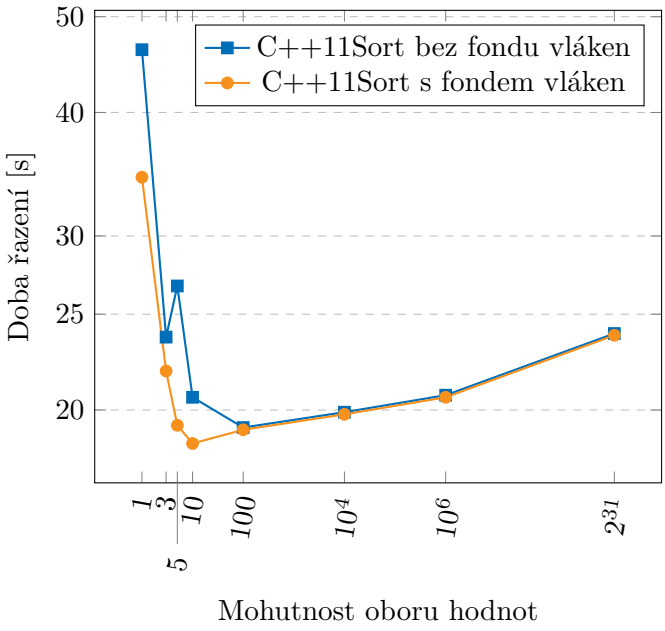
V případě opačně seřazené posloupnosti je doba řazení jen nepatrně delší než při řazení již seřazené posloupnosti. Během prvního rozdělování jsou prohozeny prakticky všechny prvky, s možnou výjimkou prostředních prvků se stejnou hodnotou. Je při tom prohazován vždy jeden prvek zprava a jeden prvek zleva. Díky tomu vzniknou po prvním rozdělování dvě již seřazené podposloupnosti. V nich není třeba dále prohazovat žádné prvky. Proto je i řazení opačně seřazené posloupnosti poměrně rychlé.

V případě řazení řetězců byly rychlosti různě seřazených vstupních dat více srovnatelné. Nejrychleji byla seřazena posloupnost náhodně uspořádaných vstupních řetězců. Naopak nejdéle trvalo řazení již seřazené posloupnosti. Řetězce se od čísel liší tím, že doba potřebná pro porovnání dvou řetězců není konstantní. Navíc kvůli proměnlivé délce řetězců je pro procesor složité předpovídat, co bude následovat po dokončení právě probíhajícího výpočtu. V případě řazení řetězců byla výrazně rychlejší verze s fondem vláken.

5.4.2 Vliv mohutnosti oboru hodnot

Měření výkonnosti algoritmu C++11Sort byla provedena i pro různé mohutnosti oborů hodnot. Prvky ve výše zmíněných řazených posloupnostech celých čísel nabývaly náhodných hodnot z rozsahu 0 až $2^{32} - 1$. Cílem experimentů v této kategorii bylo zjistit vliv mohutnosti oboru hodnot na dobu řazení. Vstupní posloupnost tedy byla generována pouze z menšího oboru hodnot.

Vliv mohutnosti oboru hodnot na dobu běhu programu je znázorněn na obrázku 5.8. Z experimentálního měření je pozorovatelný pokles doby běhu programu pro menší obory hodnot. Doba běhu programu je kratší, protože při rozdělování není třeba provádět tolik prohazování prvků. Pokud je obor hodnot naopak velmi malý, je z grafu patrný extrémní nárůst doby řazení. To je způsobeno tím, že při takto malém počtu různých hodnot je pravděpodobné, že vybraný pivot není blízký mediánu a posloupnost není rovnoměrně seřazena. Lepších výsledků by mohlo být pro malé obory hodnot dosaženo, pokud by byla posloupnost rozdělována na tři části, jak je popsáno v kapitole 1.2.6. Pokud je předem známo, že obor řazených hodnot je malý, je lepší použít pro řazení jiné řadící algoritmy, které dokážou využít tohoto faktu a které mohou pracovat i v lineárním čase. Mezi vhodnější algoritmy pro řazení posloupnosti s malou mohutností oboru hodnot patří například counting sort [26].



Obrázek 5.8: Vliv mohutnosti oboru hodnot na dobu řazení 500 milionů celých čísel.

5.5 Porovnání s existujícími implementacemi

Vytvořená implementace algoritmu C++11Sort byla následně porovnána s existujícími implementacemi paralelního quicksortu v jazyce C++, které jsou popsány v kapitole 2.3. Jmenovitě byl C++11Sort porovnáván hlavně s následujícími implementacemi:

- implementace algoritmu quicksort z Intel knihovny paralelního STL `tbb::parallel_sort` (TBB QS),
- implementace sekvenčního algoritmu quicksort z knihovny libstdc++ `std::sort` (std::sort),
- implementace algoritmu quicksort z paralelního módu libstdc++ v nevyvážené verzi `__gnu_parallel::__parallel_sort_qs` (GNU QS),
- implementace algoritmu quicksort z paralelního módu libstdc++ ve vyvážené verzi `__gnu_parallel::__parallel_sort_qsb` (GNU QSB),
- implementace algoritmu AQsort `aqsort::sort` (AQsort).

V některých testech byly požity i následující implementace:

- implementace algoritmu quicksort založená na OpenMP z knihovny Sorter Threaded `SorterThreaded<type>::sort` (STsort),

- implementace algoritmu quicksort z knihovny Parallel file quicksort `ExternalQuicksorts::quicksortMem` (PFQS),
- implementace out-of-place algoritmu quicksort Parasort `parasort` (Parasort).

Zkratky uvedené v závorkách na konci jednotlivých položek jsou dále používány ke zkrácenému označování jednotlivých implementací. C++11Sort ve verzi bez fondu vláken je dále označován jako „C++11Sort“, ve verzi s fondem vláken je označován jako „C++11Sort F“.

Při porovnávání s ostatními implementacemi byly použity nové posloupnosti, které nebyly použity k výše zmíněnému ladění parametrů. Tím bylo zabráněno případnému zkreslení výsledků, které by mohlo nastat, pokud by byly parametry algoritmu C++11Sort pouze přeučeny na určité posloupnosti používané při vývoji algoritmu.

5.5.1 Základní porovnání implementací

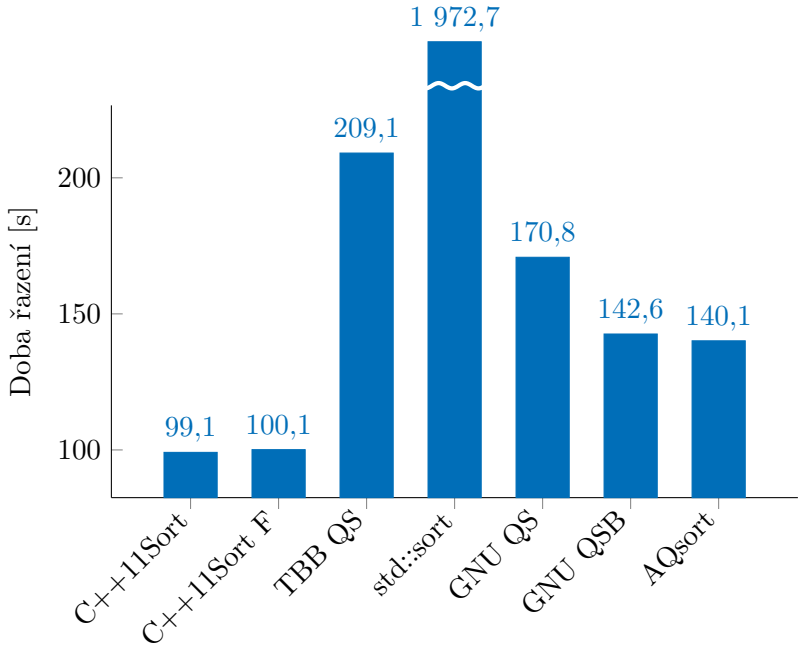
Základní porovnání implementací bylo měřeno na posloupnosti celých čísel, která obsahovala dvě miliardy prvků. Naměřené hodnoty jsou zobrazeny na obrázku 5.9. Dále proběhlo i kontrolní měření na posloupnosti, která obsahovala čtyři miliardy prvků. Výsledky byly zcela analogické, zrychlení všech implementací bylo prakticky stejné jako zrychlení při řazení posloupnosti obsahující dvě miliardy prvků.

Z porovnávaných paralelních implementací dosahovala nejhorších výsledků implementace z Intel knihovny paralelního STL. Oproti sekvenčnímu řadícímu algoritmu dosáhla pouze méně než desetinásobného zrychlení. Implementace paralelního nevyváženého řazení z knihovny libstdc++ dosáhla téměř dvanáctinásobného zrychlení. Při použití vyváženého řazení ze stejné knihovny bylo zrychlení téměř čtrnáctinásobné. Podobného zrychlení dosáhla i implementace algoritmu AQsort. Implementace vytvořené v rámci této práce byly až překvapivě rychlé. Dosahovaly shodně více než devatenáctinásobného zrychlení.

5.5.2 Testování na různých typech vstupních dat

Výkonnost jednotlivých implementací byla porovnávána pro různé typy vstupních posloupností. Byly vybrány tři základní typy posloupností, posloupnost celých čísel, posloupnost desetinných čísel a posloupnost znakových řetězců. Zkoumán byl také vliv seřazenosti vstupních dat. Byla zkoumána doba běhu implementace pro náhodně řazenou vstupní posloupnost, již seřazenou vstupní posloupnost a opačně seřazenou vstupní posloupnost.

První zkoumaná posloupnost obsahovala dvě miliardy celých čísel. Výsledky měření nad touto posloupností jsou znázorněny na obrázku 5.10. Z grafu je patrné, že pro seřazená data dosahuje zdaleka nejlepších výsledků TBB QS.



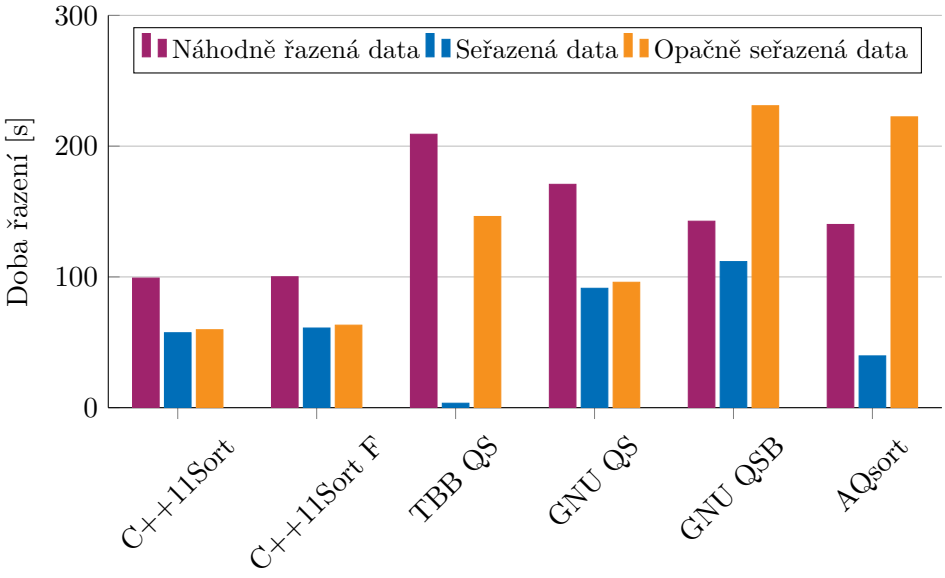
Obrázek 5.9: Porovnání doby řazení dvou miliard celých čísel různými implementacemi.

To je způsobeno tím, že na počátku kontroluje, jestli už není daná posloupnost seřazená. Výborných výsledků dosahuje také AQsort. Algoritmy z knihovny libstdc++ dosahují na již seřazených datech horších výsledků než ostatní implementace.

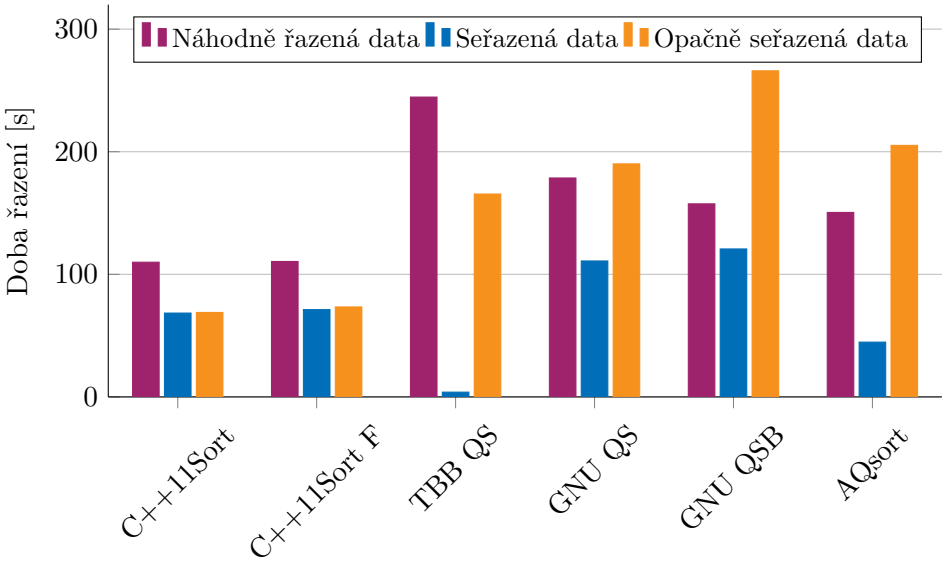
Na opačně seřazených datech dosahuje nejlepších výsledků implementace C++11Sort. Dále na těchto datech dosahuje poměrně dobrých výsledků nevyvážený quicksort z knihovny libstdc++, který dosahuje výrazně lepších výsledků oproti řazení náhodně seřazených vstupních dat. Oproti tomu vyvážený quicksort ze stejné knihovny je na opačně seřazených datech o hodně pomalejší, než při řazení náhodně řazených dat. Podobně implementace algoritmu AQsort je na opačně seřazených datech výrazně pomalejší než v případě náhodně seřazených dat.

Druhá zkoumaná posloupnost obsahovala dvě miliardy desetinných čísel. Výsledky měření nad touto posloupností jsou znázorněny na obrázku 5.11. Naměřená data byla velmi podobná naměřeným datům z řazení posloupnosti celých čísel. U všech implementací došlo k mírnému nárůstu doby nutné k seřazení posloupnosti z důvodu déletrvajícího porovnávání dvou desetinných čísel. Nejvýraznější změna nastala při řazení opačně seřazené posloupnosti implementací GNU QS. Řazení trvalo o trochu déle oproti řazení náhodně řazených vstupních dat.

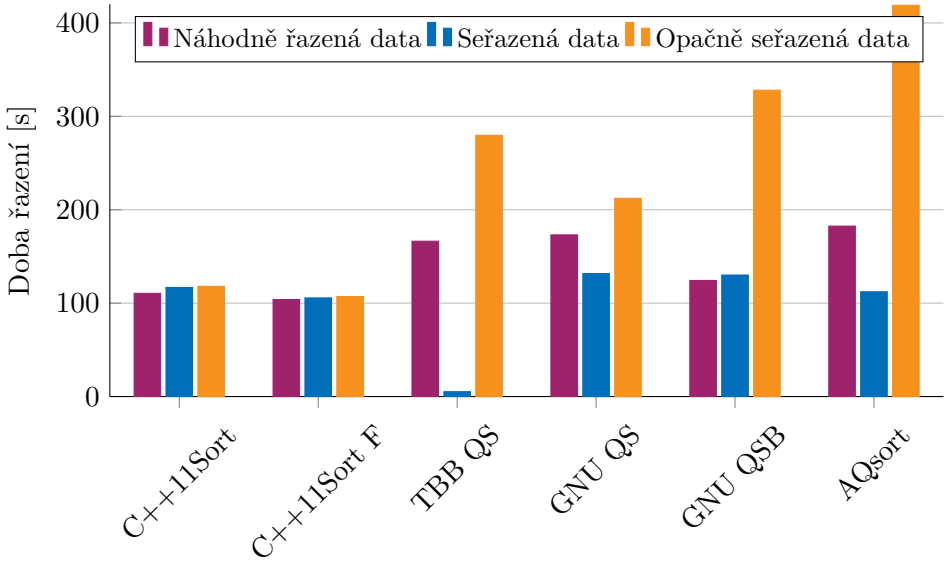
Třetí zkoumaná posloupnost obsahovala tři sta milionů znakových řetěz-



Obrázek 5.10: Vliv uspořádání dat na dobu řazení dvou miliard celých čísel jednotlivými implementacemi.



Obrázek 5.11: Vliv uspořádání dat na dobu řazení dvou miliard desetinných čísel jednotlivými implementacemi.



Obrázek 5.12: Vliv uspořádání dat na dobu řazení tří set milionů znakových řetězců jednotlivými implementacemi.

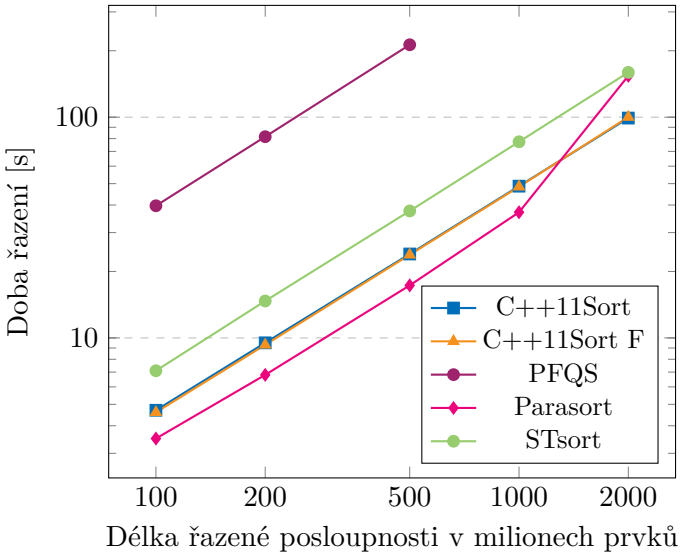
ců o délce pět až třicet znaků. Výsledky měření nad touto posloupností jsou znázorněny na obrázku 5.12. Struktura naměřených dat je značně rozdílná oproti datům naměřeným při řazení číselných posloupností. Při řazení řetězců trvá porovnávání jednotlivých dvojic prvků déle. Nejrychlejší implementací je nadále C++11Sort, velmi dobrých výsledků dosahuje také vyvážená verze GNU QSB. Zbylé tři algoritmy jsou podobně rychlé. TBB QS dosahuje, na rozdíl od předchozích měření, výrazně pomalejšího času pro řazení opačně seřazených dat. GNU QSB dosahuje na seřazených datech mírně horšího výsledku než na datech náhodně řazených, přičemž u řazení čísel tomu bylo naopak.

5.5.3 Výsledky experimentů s dalšími implementacemi

Pro zajímavost jsou zde uvedeny i výsledky experimentální měření dalších implementací, které byly zmíněny v kapitole 2.3. Naměřené výsledky jsou znázorněny na obrázku 5.13.

Implementace PFQS založená na C++11 vláknech dosahovala přibližně sedmkrát horších výsledků než konkurenční implementace. Horší výsledky byly očekávané, vzhledem k pouze naivní implementaci rozdělování.

Parasort, který je však out-of-place, dosahoval výborných výsledků, pokud se řazená data vešla do paměti. Dosahoval přibližně poloviční časů oproti konkurenčním implementacím, proto je určitě zajímavý v případech, kdy je k dispozici dostatek paměti. STsort byl v testech přibližně o 50 % pomalejší



Obrázek 5.13: Vliv délky posloupnosti celých čísel na dobu běhu jejího řazení pro další implementace.

než C++11Sort.

5.5.4 Řazení malých posloupností

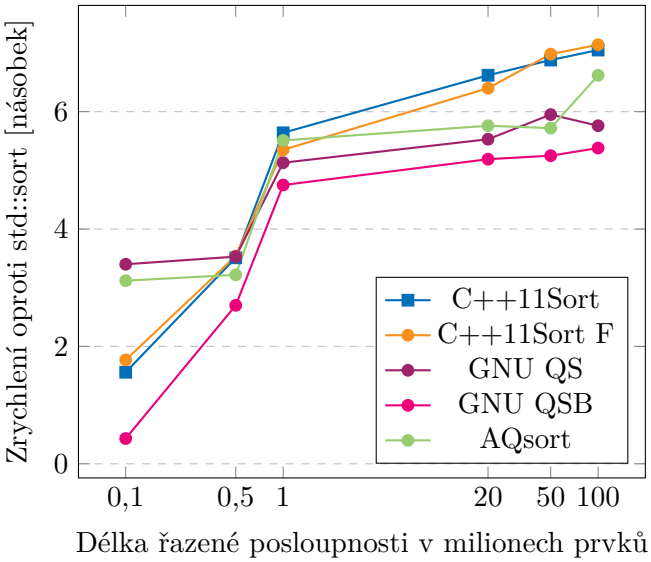
Cílem dalšího měření bylo zjistit, jakých výsledků dosahuje C++11Sort na menších datech a v prostředí s méně vlákny v porovnání s ostatními používanými implementacemi. Měření proběhlo s využitím osmi jader na posloupnostech celých čísel. Výsledky měření jsou znázorněny na obrázku 5.14.

Z naměřených výsledků vyplývá, že GNU QSB je při řazení posloupnosti obsahující sto tisíc prvků na osmi jádrech pomalejší než sekvenční řadící algoritmus. Ostatní implementace jsou i pro takto malou řazenou posloupnost rychlejší. Nicméně smysluplnějšího využití všech jader je dosaženo až při řazení několika desítek milionů prvků.

5.5.5 Řazení maticových elementů

Cílem posledního měření bylo zjistit využitelnost algoritmu C++11Sort při řazení maticových elementů. Řídké matice je někdy užitečné rozdělit do bloků určité velikosti. Tento problém v sobě zahrnuje řazení nenulových maticových prvků podle klíče, který jednotlivé elementy seskupuje do bloků [27].

Řazené matice byly získány z veřejně dostupné kolekce SuiteSparse Matrix Collection [28]. Matice byly v těchto testech uloženy ve formátu COO (coordinate storage format) [29]. Tedy byly zadány ve třech polích, které obsahovaly řádkové indexy, sloupcové indexy a samotné hodnoty nenulových prvků matice. Z toho důvodu bylo nutné vytvořený C++11Sort mírně upravit tak, aby

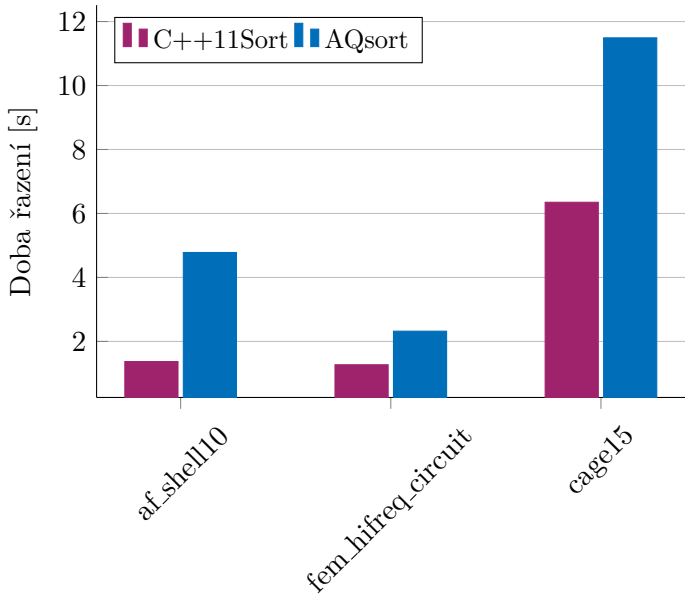


Obrázek 5.14: Zrychlení při řazení menších posloupností na osmi jádrech různými implementacemi. Graf vyjadřuje kolikrát byla implementace rychlejší oproti sekvenčnímu řadícímu algoritmu `std::sort`.

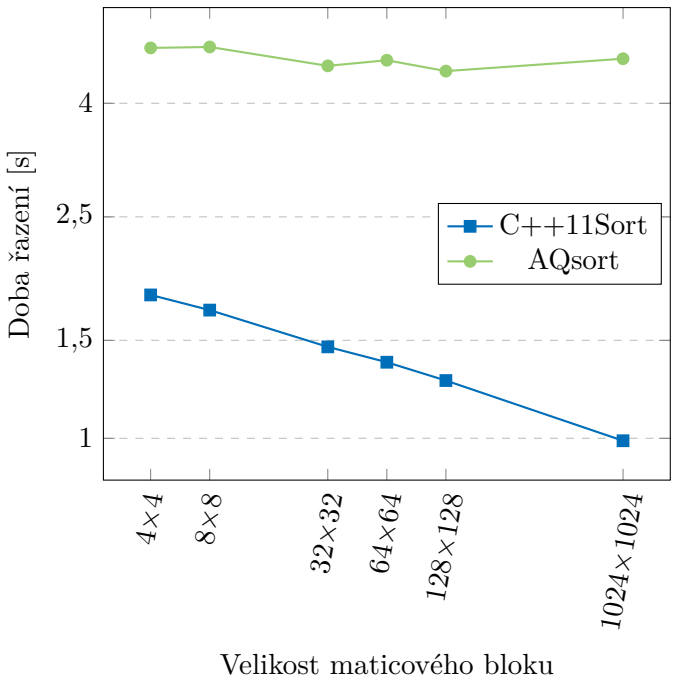
dokázal řadit více polí najednou, podobně jako je tomu u výše popsaného algoritmu AQsort. Prakticky úprava algoritmu spočívala v tom, že samotný řadící algoritmus nepracuje s konkrétními prvky pole, ale pouze s indexy multielementů. Do algoritmu jsou zároveň předávány ukazatele na funkce, které slouží k porovnání a prohození dvou multielementů na základě jejich indexů.

Měření byla provedena na třech maticích, které jsou ve výše zmíněné kolekci dohledatelné pod názvy af_shell10, cage15 a fem_hifreq_circuit. Doba řazení maticových multielementů algoritmy C++11Sort a AQsort je znázorněna na obrázku 5.15. Na obrázku 5.16 je znázorněn vliv velikosti maticových bloků na dobu řazení matice af_shell10.

Na pozorovaných maticích dosahoval C++11Sort o něco lepších výsledků oproti algoritmu AQsort. Například při použití maticových bloků o velikosti 64×64 řadil C++11Sort nenulové prvky matic přibližně dvojnásobně rychleji než AQsort. Z měření je patrné i odlišné chování obou řadících algoritmů pro různé velikosti maticových bloků. C++11Sort dosahoval výrazně lepších výsledků, pokud byla zvolena velká velikost bloku. Oproti tomu jsou výsledky AQsortu na různě velkých blocích více vyrovnané.



Obrázek 5.15: Porovnání doby řazení multielementů různých matic při použití bloků o velikosti 64×64.



Obrázek 5.16: Vliv velikosti maticového bloku na dobu řazení multielementů matice af_shell10.

Závěr

Tato práce se zabývá návrhem a efektivní implementací paralelní verze algoritmu quicksort, která je implementovatelná pouze na základě C++11 vláken a příslušných synchronizačních technik, tj. bez použití nadstandardních rozšíření jazyka C++ a externích knihoven.

Byla provedena rešerše existujících paralelních verzí algoritmu quicksort dostupných pro jazyk C++. Byly prozkoumány i algoritmy pro rozdělování a jejich implementace. Existující verze řadících a rozdělovacích algoritmů byly podrobně popsány. Během rešerše nebyla nalezena efektivní implementace algoritmu quicksort, která by využívala pouze standardních knihoven jazyka C++.

Na základě provedené rešerše byla navržena a implementována efektivní paralelní verze algoritmu quicksort. Vytvořený algoritmus, který byl nazván C++11Sort, je založen na C++11 vláknech a je implementován bez použití nadstandardních rozšíření jazyka C++ a externích knihoven. Byly vytvořeny dvě verze algoritmu, přičemž jedna z nich používá fond vláken. Důležitou součástí vytvořeného algoritmu je i efektivní rozdělování posloupnosti, jehož implementace je také založena na C++11 vláknech. Rozdělování je možné využít i samostatně v jiných aplikacích.

Následně byla provedena na různých datech řada měření, která testovala vytvořenou implementaci. Provedeno bylo také experimentální porovnání s ostatními existujícími implementacemi. C++11Sort dosahoval v porovnání s ostatními implementacemi až překvapivě dobrých výsledků. Například při řazení dvou miliard celých čísel byl C++11Sort rychlejší o 28 % oproti nejrychlejší konkurenční implementaci.

V budoucnu by v návaznosti na tuto práci bylo možné vytvořit i další paralelní verze algoritmů, například verzi paralelního out-of-place řadícího algoritmu merge sort. Zajímavé by také bylo provést měření na různých typech architektur.

Literatura

[1] Martin Mareš, T. V.: *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z.s.p.o., první vydání, 2007, ISBN 978-80-88168-19-5.

[2] Mohammed, A.; Othman, M.: Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm. *Information Technology Journal*, ročník 6, č. 3, 2007: s. 424–427, doi:10.3923/itj.2007.424.427.

[3] LaMarca, R. E., Anthony; Ladner: The Influence of Caches on the Performance of Sorting. *Journal of Algorithms*, ročník 31, č. 1, 1999: s. 66–104.

[4] Langr, D.; Tvrdík, P.; Šimeček, I.: AQsort: Scalable Multi-Array In-Place Sorting with OpenMP. *Scalable Computing: Practice and Experience*, ročník 17, 10 2016.

[5] Singler, J.; Konsik, B.: The GNU Libstdc++ Parallel Mode: Software Engineering Considerations. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-031-9, s. 15–22. Dostupné z: `http://doi.acm.org/10.1145/1370082.1370089`

[6] Thread support library. 2016. Dostupné z: `https://en.cppreference.com/w/cpp/thread`

[7] Lee, E.: The Problem with Threads. *Computer*, ročník 39, č. 5, 2006: s. 33–42, doi:10.1109/mc.2006.180.

[8] Hoberock, J.: The Parallelism TS Should be Standardized. 2016. Dostupné z: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0024r2.html`

[9] Chapman, B.; Jost, G.; van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). 01 2007.

[10] Alexey, M.: Get Started with Parallel STL. 2018. Dostupné z: https://software.intel.com/en-us/articles/get-started-with-parallel-stl

[11] Kukanov, A.: Proposal to contribute Intel's implementation of C++17 parallel algorithms. 2017. Dostupné z: https://gcc.gnu.org/ml/libstdc++/2017-11/msg00112.html

[12] Filipek, B.: How to Boost Performance with Intel Parallel STL and C++17 Parallel Algorithms. 2018. Dostupné z: https://www.bfilipek.com/2018/11/pstl.html

[13] Contreras, G.; Martonosi, M.: 1 Characterizing and Improving the Performance of Intel Threading Building Blocks. 10 2008, s. 57 – 66.

[14] Cantalupo, C.: Sorter Threaded. 2014. Dostupné z: https://github.com/cmcantalupo/sorter_threaded

[15] Putze, F.; Sanders, P.; Singler, J.: MCSTL: The Multi-core Standard Template Library. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-602-8, s. 144–145. Dostupné z: http://doi.acm.org/10.1145/1229428.1229458

[16] Tsigas, P.; Zhang, Y.: A Simple, Fast Parallel Implementation of Quicksort And Its Performance Evaluation on SUN Enterprise 10000. 03 2003, ISBN 0-7695-1875-3, s. 372– 381.

[17] O Neal, B.: Using C++17 Parallel Algorithms for Better Performance. 2018. Dostupné z: https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/

[18] Parallel File Quicksort. 2019. Dostupné z: https://github.com/kbuci/parallel-file-quicksort

[19] Baserinia, A.: C++ Parallel Sort. 2016. Dostupné z: https://github.com/baserinia/parallel-sort

[20] Wild, S.: *Java 7 Dual-Pivot Quicksort*. AV Akademikerverlag, 2013.

[21] Francis, R.; Pannan, L.: A parallel partition for enhanced parallel QuickSort. *Parallel Computing*, ročník 18, 05 1992: s. 543–550.

[22] Frias, L.; Petit, J.: Parallel Partition Revisited. 05 2008, s. 142–153.

[23] Frias, L.; Petit, J.: Presentation of Parallel Partition Revisited. 2008. Dostupné z: https://pdfs.semanticscholar.org/presentation/ce3a/16c05ab1464821a5cfbfc91ee12a5f6cdbe8.pdf

[24] Singler, J.: Presentation of The GNU libstdc++ parallel mode:Benefit from Multi-Core using the STL. Dostupné z: https://ls11-www.cs.tu-dortmund.de/people/gutweng/AD08/VO11_parallel_mode_overview.pdf

[25] Launch Thread performance. 2017. Dostupné z: https://bitbucket.org/omnifarious/launch_thread_performance/src/default/

[26] Bajpai, K.; Kots, A.: Implementing and Analyzing an Efficient Version of Counting Sort (E-Counting Sort). *International Journal of Computer Applications*, ročník 98, č. 9, 2014: s. 1–2, doi:10.5120/17208-7427.

[27] Langr, D.; Šimeček, I.; Dytrych, T.: Block iterators for sparse matrices. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2016, s. 695–704.

[28] SuiteSparse Matrix Collection. 2019. Dostupné z: https://sparse.tamu.edu/

[29] Saad, Y.: *Iterative Methods for Sparse Linear Systems*. 01 2003.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

F&A fetch-and-add

GCC GNU Compiler Collection

GNU GNU's Not Unix

LLVM Low Level Virtual Machine

MCSTL Multi-Core Standard Template Library

MSVC Microsoft Visual C++

OpenMP Open Multi-Processing

PFQS Parallel file quicksort

POSIX Portable Operating System Interface

PPL Parallel Patterns Library

QSB quicksort balanced

QS quicksort

RAI Random Access Iterator

STL Standard Template Library

ST Sorter Threaded

TBB Threading Building Blocks

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Schovankova_Klara_2019.pdf` ..... text práce ve formátu PDF