Zadání bakalářské práce

Název:	Porovnání vícevláknových implementací
Student:	Jan Nyklíček
Vedoucí:	Ing. Michal Šoch, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s optimalizační úlohou "0-1 Batoh", jejímž cílem je umístění podmnožiny předmětů (předmět má specifikován svoji cenu a váhu) do batohu omezené kapacity tak, aby cena nákladu byla maximální.

Danou úlohu naimplementujte v jazyce C/C++ jako sekvenční aplikaci využívající hrubou sílu.

Seznamte se také s možnostmi vícevláknového programování, konkrétně s POSIXovými vlákny, C++ vlákny a OpenMP.

Následně tuto sekvenční aplikaci paralelizujte pomocí těchto tří různých vícevláknových modelů.

Vzniklé tři vícevláknové aplikace porovnejte vzájemně mezi sebou z hlediska zrychlení měřením na vícejádrovém systému. Dále vyhodnoťte náročnost vícevláknové implementace z hlediska času a obtížnosti (tj. kolik času bylo potřeba na modifikaci sekvenčního kódu na vícevláknový s použitím dané technologie a jak to bylo náročné naprogramovat a odladit). Pro tyto účely provádějte evidencí spotřebovaného času.

Aplikace řádně otestujte.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 27. září 2022 v Praze.

Bakalářská práce

POROVNÁNÍ VÍCEVLÁKNOVÝCH IMPLEMENTACÍ

Jan Nyklíček

Fakulta informačních technologií
Katedra teoretické informatiky
Vedoucí: doc. Michal Šoch, Ph.D.
11. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Jan Nyklíček. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Naklíček Jan. *Porovnání vícevláknových implementací*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování **vii**

Prohlášení **viii**

Abstrakt **ix**

Úvod **1**

1 Úvod do paralelizmu **2**
1.1 Procesy a vlákna 2
1.2 Nedeterminismus 2
1.3 Zrychlení a efektivita 3

2 Modely paralelizace v C++ **4**
2.1 POSIX vlákna 4
2.2 C++ vlákna 4
2.3 OpenMP 4

3 Problém batohu 0-1 **6**
3.1 Popis problému batohu 0-1 6
3.2 formální definice 6

4 Analýza a návrh **7**
4.1 Návrh tříd 7
4.2 Reprezentace instance problému batohu 7
4.3 Algoritmus sekvenčního řešení 7
4.4 Paralelizace 8
4.4.1 Návrh algoritmu vhodného k paralelizaci 8
4.4.2 Paralelizace algoritmu 9

5 Implementace sekvenční aplikace **10**

6 Implementace vícevláknových aplikací **12**
6.1 implementace společné části vícevláknových aplikací 12
6.2 Implementace POSIXovými vlákny 15
6.3 Implementace C++ vlákny 18
6.4 Implementace pomocí OpenMP 19

7 Testování aplikací **21**
7.1 Korektnost řešení 21
7.2 Paralelizace 21

8 Používání Aplikací **22**

9 Náročnost implemetace vícevláknových aplikací **23**

10 Testování zrychlení **24**
10.1 Metodika měření 24
10.2 Výsledky měření 24
10.3 Zhodnocení výsledků 25

Závěr **27**

Literatura **27**

A Obsah přiloženého média **29**

Seznam obrázků

4.1 Vývojový diagram pracovního vlákna 9

8.1 Ukázka použití vícevláknové aplikace 22

10.1 Průměrná efektivita v závislosti na počtu vláken 26
10.2 Průměrné zrychlení v závislosti na počtu vláken 26

Seznam tabulek

9.1 Tabulka časové náročnosti implementace vícevláknových aplikací 23

10.1 Tabulka výsledků měření sekvenční aplikace 25
10.2 Tabulka výsledků měření vícevláknové aplikace implementované POSIX vlákny . 25
10.3 Tabulka výsledků měření vícevláknové aplikace implementované C++ vlákny . . 25
10.4 Tabulka výsledků měření vícevláknové aplikace implementované OpenMP 25

Seznam výpisů kódu

5.1 Třída KnapsackSolver 10
5.2 metoda solve řešící problém batohu 11
6.1 Abstraktní třída MultithreadedKnapsackSolver 13
6.2 metoda getFixedMask 14
6.3 Metoda setBestResult 14
6.4 metoda setFixedMask 14
6.5 Metoda knapsackSolve 15
6.6 Třída POSIXKnapsackSolver 16
6.7 Metoda solve implementována POSIXovými vlákny 17
6.8 statická metoda threadHelper 17
6.9 Metoda thread implementovaná POSIXovými vlákny 18
6.10 Třída CPPKnapsackSolver 18
6.11 Metoda solve implementovaná pomocí C++ vláken 19
6.12 Metoda thread implementovaná pomocí C++ vláken 19

6.13 Třída OMPKnapsackSolver 20
6.14 Metoda solve implementována pomocí OpenMP 20

List of Algorithms

1 Pseudokód řešení problému batohu 0-1 hrubou silou 8
2 Pseudokód řešení problému batohu 0-1 hrubou silou, vhodného k paralelizaci . . 9

Chtěl bych poděkovat Ing. Michal Šoch, Ph.D. za odborné vedení bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2023 ....

Abstrakt

Tato práce popisuje implementaci sekvenční aplikace řešící problém batohu 0-1 v jazyce C++ a následnou paralelizaci pomocí Posix vláken, C++ vláken, a OpenMP. Výsledné tři vícevláknové aplikace analyzuje z hlediska náročnosti na implementaci a dosaženého zrychlení na vícejádrovém systému.

Klíčová slova paralelní programování, C++ vlákna, POSIX vlákna, OpenMP, problém batohu 0-1

Abstract

This thesis describes the implementation of a sequential application for solving the 0-1 knapsack problem in C++, followed by parallelization using Posix threads, C++ threads, and OpenMP. The resulting three multi-threaded applications are analyzed in terms of implementation complexity and achieved speedup on a multi-core system.

Keywords parallel programming, C++ threads, POSIX threads, OpenMP, knapsack problem 0-1

Úvod

Vývoj mikroprocesorů již nedosahuje takového zvýšení výkonu jako dříve. Důsledkem tohoto se návrh procesorů dal směrem paralelizace a výrobci začali navrhovat procesory s větším počtem jader. U většiny sekvenčních programů ale přidáním více jader zrychlení nedocílíme, jelikož nebyly navrhnuté tak aby věděli jak práci rozdělit. Paralelní programování umožňuje rozdělit problém na části tak aby bylo možné ho vykonávat současně na několika výpočetních prostředcích. [1]

V C++ existuje několik modelů pro paralelizaci programů běžících na systémech se sdílenou pamětí. Mezi nejpopulárnějšimi z nich patří C++ vlákna, POSIXová vlákna a OpenMP. Každý z těchto modelů poskytuje jinou úroveň kontroly nad vlákny. V rámci této práce byla implementována sekvenční verze aplikace řešící problém batohu 0-1 hrubou silou. Následně jsou tyto modely použity pro paralelizaci sekvenční aplikace.

Výsledné vícevláknové aplikace jsou porovnány zhlediska dosaženého zrychlení na vícejádrovém systému a času spotřebovaného na implentaci a odladění.

Kapitola 1
Úvod do paralelizmu

Tato kapitola slouží jako úvod do paralelizmu. Autor v této kapitole vychází z [1].

1.1 Procesy a vlákna

Operační systém je zodpovědný za správu hardwarových a softwarových zdrojů počítače, včetně rozhodování o tom, které programy mohou běžet a ovládání alokace paměti. Když uživatel spustí program, operační systém vytvoří proces, což je instance programu, který se vykonává. Proces se skládá ze spustitelného strojového kódu, virtuální paměti, deskriptorů zdrojů, bezpečnostních informací a stavu procesu. Moderní operační systémy umožňují multitasking, což znamená podporu zdánlivě současného spuštění více programů i na jednojádrových systémech, jelikož jeden proces beží po dobu krátkého časového úseku, po kterém operační systém může spustit jiný proces.

Vlákno (anglicky thread) je základní jednotka procesu, která umožňuje víceúlohové (paralelní) zpracování dat a činností v rámci jednoho programu. Vlákna v rámci jednoho procesu sdílí většinu prostředků daného procesu, jako například spustitelný zdrojový kód nebo virtuální paměť. Jednotlivá vlákna naopak mají svůj vlastní zásobník a čítač instrukcí, aby mohly vykonávat kód nezávisle na sobě. Díky tomuto jsou "lehčí"než procesy a na většině systémech lze vytvářet a přepínat mezi vlákny mnohem rychleji.

1.2 Nedeterminismus

V systémech typu MIMD, kde procesory vykonávájí instrukce asynchroně, může nastat nedeterminismus. což znamená že stejný vstup může vést k různým výstupům. Pokud více vláken přistupuje najednou ke sdílené paměti, může dojít k nekonzistentnosti v závislosti na vzájemném pořadí vykonání instrukcí těchto vláken. Takovéto chyby se nazývají **race condition**. Způsob, jakým se těmto chybám vyhnout, je zajistit že část kódu, ve které by se mohly vyskytnou, jsou vykonávány najednou pouze jedním vláknem. Tyto sekce kódu se nazývají **kritické sekce**. Vzhledem že kritické sekce serializují program, je žádoucí aby se v kódu vyskytovaly co nejméně.

Jedním z hlavních mechanizmů jak zajistit výlučný přístup vlákna ke kritické sekci jsou **mutexy** (mutual exclusion lock). Mutex je objekt který vyžaduje podporu v hardwaru. Hlavní myšlenkou mutexu je, že chrání kritickou sekci. Pokud nějaká výpočetní jednotka chce provádět kritickou sekci, musí nejprve získat mutex a po dokončení jej znovu uvolní. Pokud se mutex, který získala jedna výpočetní jednotka, pokusí získat další výpočetní jednotka, je blokována dokud se mutex neuvolní.

1.3 Zrychlení a efektivita

Mezi metriky hodnocení kvality paralelizace algoritmu patří zrychlení a efektivita.

Zrychlení: Zrychlení porovnává čas běhu sekvenčního programu ku době běhu paralelního programu a je vyjádřeno vzorcem $Z = T_{sekvenční}/T_{paralelní}$. Ideální dosažitelná hodnota je p kde p je počet použitých výpočetních jednotek a představuje lineární zrychlení.

Efektivita: Efektivita představuje průměrný přínos jedné výpočetní jednotky na zrychlení a je definován vzorcem $E = Z/p$. Ideální dosažitelná hodnota je 1 (100%) odpovídající lineárnímu zrychlení.

Lineární zrychlení je ve většině případů nedosažitelné, jelikož samotná správa jednotlivých vláken přidává práci navíc. Dalšími faktory které zrychlení snižují jsou kritické sekce a práce spojená s přerozdělováním práce mezi jednotlivá vlákna. Navíc s rostoucím počtem jader se efektivita dále snižuje.

Kapitola 2

Modely paralelizace v C++

Tato kapitola popisuje použité modely pro vývoj vícevláknových aplikací v jazyce C++.

2.1 POSIX vlákna

Součástí standardu POSIX je rozhraní pro práci s vlákny Pthread API. Procedury, které tvoří rozhraní Pthreads API, lze neformálně rozdělit do čtyř hlavních skupin:

Správa vláken: vytváření, oddělování, spojování vláken

Mutexy: vytváření, rušení, zamykání a odemykání mutexů.

Podmínkové proměnné: Procedury, které se zabývají komunikací mezi vlákny.

Synchronizace: Procedury, které spravují zámky čtení a zápisu a bariéry.

[3]

2.2 C++ vlákna

C++11 Thread Support Library je knihovna jazyka C++, která poskytuje prostředky pro paralelní programování. Tato knihovna byla přidána do standardu jazyka C++ od verze C++11. Tato knihovna obsahuje třídy a funkce pro práci s vlákny , atomickými operacemi, synchronizací, mutexy, podmíněnými proměnnými a dalšími nástroji pro paralelní programování. [4]

2.3 OpenMP

OpenMP (Open Multi-Processing) je vysokoúrovňové API pro paralelní programování nad sdílenou pamětí v jazyce C, C++ a Fortran. Skládá se ze tří hlavních komponent:

- Direktiva pro kompilátor
- Proměnné OpenMP prostřední
- knihovna funkcí běhového prostředí OpenMP.

OpenMP používá tzv. ”fork-join”model paralelního vykonávání. Všechny OpenMP programy začínají jako jeden proces (hlavní vlákno). Hlavní vlákno se vykonává sekvenčně, dokud nenarazí na první konstrukci paralelního regionu, vytvoří tým vláken. Příkazy v programu které jsou uzavřeny v konstrukci paralelního regionu, jsou vykonávány paralelně mezi různými vlákny týmu. Když vlákna týmu dokončí příkazy v konstrukci paralelního regionu, synchronizují se a ukončí. Dále zůstává pouze hlavní vlákno. [5]

Kapitola 3

Problém batohu 0-1

3.1 Popis problému batohu 0-1

Problém batohu 0-1 je NP-úplný optimalizační problém. Neformálně ho lze definovat následovně:

Mějme batoh o dané kapacitě a seznam předmětů s jejich cenami a vahami. Cílem je vybrat podmnožinu předmětů která bude mít největší celkovou cenu ale nepřesáhne svou celkovou vahou kapacitu batohu. [2]

3.2 formální definice

Problém batohu 0-1 je: mějme množinu n předmětů a batoh, kde

w_i - váha i-tého předmětu ($i = 1, 2, ..., n$)

v_i - cena i-tého předmětu ($i = 1, 2, ..., n$)

c - maximální kapacita batohu

Najděte podmnožinu předmětů takovou, že:

$$\text{maximalizujte } \sum_{j=1}^{n} v_j x_j$$

$$\text{za podmínky } \sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0, 1\},\ j \in \{1, 2, ..., n\}$$

$$\text{kde } x_j = \begin{cases} 0 & j\text{-tý předmět není v batohu} \\ 1 & j\text{-tý předmět je v batohu} \end{cases}$$

[2]

Kapitola 4
Analýza a návrh

4.1 Návrh tříd

Aplikace budou implementovány třídou, která bude mít metody pro získání instance problému batohu ze standardního vstupu, vypsání výsledků na standardní výstup a metodu která provede samotné nalezení řešení. Informace o instanci batohu a řešení budou uloženy v rámci atributů této třídy.

4.2 Reprezentace instance problému batohu

Instance problému batohu je tvořena kapacitou batohu a předměty se svými atributy. Předměty jsou reprezentovány polem dvojic hodnot délky stejné jako počet předmětů, kde první hodnota i-tého prvku v poli reprezentuje cenu i-tého předmětu a druhá jeho váhu. Definice problému batohu striktně neurčují zda-li atributy předmětů a kapacita batohu musí mít nezápornou hodnotu. Vzhledem k tradičním použitím problému batohu budou uvažovány pouze nezáporné hodnoty. Atributy předmětů jsou reprezentovány datovým typem *unsigned int* a maximální kapacita či sumy atributů předmětů v batohu *unsigned long*.

Kombinace předmětů v batohu je reprezentována polem bool hodnot o velikosti počtu předmětů kde, i-tý prvek pole udává, zdali je i-tý předmět vložen do batohu (hodnota true), nebo není (hodnota false).

4.3 Algoritmus sekvenčního řešení

Pseudokód Algoritmu 1 popisuje algoritmus řešení problému batohu 0-1 hrubou silou. Tento algoritmus iterativně zkouší všechny možné kombinace předmětů v batohu. Pro každou kombinaci předmětů je výpočítána celková cena a váha předmětů v batohu a následně je zkontrolována, zdali splňuje váhový limit batohu a pokud je aktuální kombinace nejlepší dosažená, je uchována.

Iterace přes variace binárních sekvencí délky n je realizována principem binárního čítače. Pro získání následující kombinace se k první hodnotě pole přičte hodnota 1. Toto odpovídá operaci inverze v binární soustavě. Pokud dojde k přetečení prvku v poli (původní hodnota je 1), přenáší se hodnota 1 na následující prvek ve vektoru. Pokud k přetečení dojde i na posledním prvku, znamená to, že byly vyčerpány všechny kombinace, a pole se vrátí na počáteční kombinaci, kdy v batohu nejsou žádné předměty.

Algorithmus 1 Pseudokód řešení problému batohu 0-1 hrubou silou

Input: n, c, $v_i i = 1^n$, $w_i i = 1^n$
$bestValue \leftarrow 0$
$bestCombination \leftarrow 0$
for all binary sequences x of length n **do**
  $value \leftarrow 0$
  $weight \leftarrow 0$
  for $i = 1$ **to** n **do**
    if $x_i = 1$ **then**
      $value \leftarrow value + v_i$
      $weight \leftarrow weight + w_i$
    end if
  end for
  if $weight \leq c$ and $value > bestValue$ **then**
    $bestValue \leftarrow value$
    $bestCombination \leftarrow x$
  end if
end for
Output: $bestValue$, $bestCombination$

4.4 Paralelizace

Pro paralelizaci programu je potřeba vhodně rozdělovat práci mezi vlákna. Je tedy třeba si řešený problém rozdělit na vhodný počet částí, které budou přiřazovány jednotlivým vláknum k řešení. Je také dobré minimalizovat závislost jednoho podproblému na jiných tak aby vlákna vstupovala co nejméně často do kritických sekcí. Následující sekce popisuje principy, použité v implementaci všech 3 vícevláknových aplikací.

4.4.1 Návrh algoritmu vhodného k paralelizaci

Množinu všech kombinací binární masky reprezentující kombinaci n předmětů lze rozdělit na 2^d stejně velkých množin, kde každá množina obsahuje kombinace masky která má prvních d hodnot nastavených vždy na stejnou hodnotu. Takto lze rozdělit práci mezi jednotlivá vlákna kdy množina kombinací předmětů batohu které má vlákno řešit je jednoznačně určena maskou prvních d prvků (dále pouze fixní maska).

Množství vytvořených podproblémů se odvíjí od velikosti fixní masky, která je závislá na počtu vláken. Pokud chceme aby každé vlákno získalo alespoň jednu variaci fixní masky, je třeba zvolit masku délky alespoň $\lceil \log_2(počet_vláken) \rceil$. Pokud počet vláken bude rovný nějaké mocnině 2, každé vlákno dostane přesně jednu kombinaci fixní masky. V jiných případech ale dojde k situaci kdy bude práce nerovnoměrně rozdělená kdy některá vlákna by dostali 1 a jiné dvě fixní masky. Aby mohlo dojít k více rovnoměrnému rozložení podproblému batohu mezi vlákna, je k výše zmíněnému výrazu přičten ještě $\lfloor \log_2(počet_zbývajících_předmětů) \rfloor$.

Výsledný algoritmus 2 je tvořen cyklem, ve kterém se na základě fixní masky, zkouší pouze různé kombinace předmětů, ke kterým se fixní maska nevztahuje. Následně je aktualizován nejlepší dosažený výsledek.

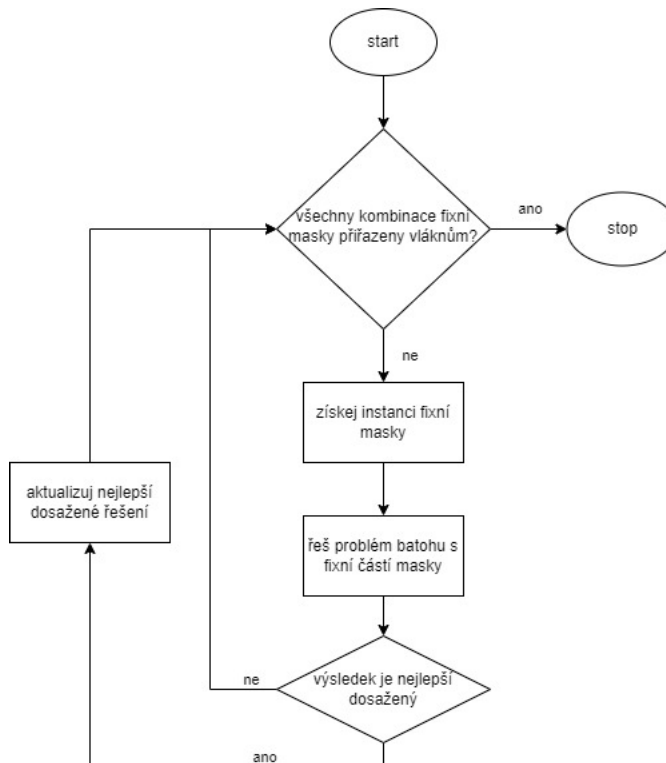
Algorithmus 2 Pseudokód řešení problému batohu 0-1 hrubou silou, vhodného k paralelizaci

Input: d, n, c, $v_i i = 1^n$, $w_i i = 1^n$
$globalBestValue \leftarrow 0$
$globalBestCombination \leftarrow 0$
for all binary sequences $fixedMask$ of length d **do**
  $bestValue, bestCombination \leftarrow test_all_combinations_with_fixed_mask()$
  if $BestValue > globalBestValue$ **then**
    $globalBestValue \leftarrow value$
    $globalBestCombination \leftarrow x +$ bestCombination
  end if
end for
Output: $GlobalBestValue$, $GlobalBestCombination$

4.4.2 Paralelizace algoritmu

Vzhledem k tomu, že algoritmus je tvořen cyklem, kde každá iterace je závislá na předchozí kvůli výpočtu variace fixní masky, je použit model do-across paralelismu, kde části iterací cyklu, které jsou na sobě závislé jsou prováděny sekvenčně a paralelně jsou prováděny části iterace cyklu, které jsou vzájemně nezávislé. Získání fixní části masky je závislé na předchozí iteraci. Tato operace je proto prováděna v kritické sekci sekvenčně. Stejně tak je prováděn i kontrola a aktualizace dílčích výsledků. Vzhledem k náročnosti výpočtu následující kombinace fixní masky a porovnávání dílčích výsledků ku samotnému zkoušení různých kombinací předmětu, bude paralelizovatelná část cyklu tvořit většinovou část. Obrázek 4.1 zobrazuje chování jednotlivého vlákna provádějící tento cyklus.

■ **Obrázek 4.1** Vývojový diagram pracovního vlákna



Kapitola 5

Implementace sekvenční aplikace

K řešení jednotlivých instancí problém batohu využívá sekvenční aplikace třídu *KnapsackSolver*, popsanou ve výpisu kódu 5.1. Informace o problému batohu lze předat třídě v konstruktoru z důvodu testování nebo metodou *loadFromStdIn* která načte informace ze standartního vstupu používanou samotnou aplikací. Nalezení řešení problému batohu proběhne zavoláním metody *solve*. Výsledek lze získat metodou *getResult* a nebo rovnou vypsat na standartní výstup metodou *printResult*.

Výpis kódu 5.1 Třída KnapsackSolver

```
class KnapsackSolver {
private:
    std::vector<std::pair<unsigned int, unsigned int>> items;
    unsigned long maxWeight;
    unsigned long bestValue;
    std::vector<bool> bestItemMask;
    bool solved;
public:
    KnapsackSolver();
    KnapsackSolver(std::vector<std::pair<unsigned int, unsigned int>> &
        items, unsigned long maxWeight);
    bool loadFromStdIn();
    unsigned long getResult(std::vector<bool> & bestItemMask);
    void printResult();
    void solve();
};
```

V metodě *solve*, zobrazené ve výpisu kódu 5.2, je použita hrubá sila pro řešení problému batohu. Nejprve je vytvořen vektor itemMask reprezentující kombinaci předmětů, která obsahuje pouze hodnoty false. Tato kombinace odpovídá počátečnímu stavu, kdy je batoh prázdný a celková hodnota předmětů v něm je nulová.

Na rozdíl od algoritmu popsaného v sekci 4.3, kde se cena a váha předmětů v batohu počítají znovu po vytvoření každé masky, v této implementaci se celková cena a váha předmětů vypočítávají současně s modifikací masky. Počáteční maska neobsahuje žádné předměty, takže celková cena a váha jsou nulové. Pokud se mění i-tá hodnota v masce, zároveň se přičte či odečte od celkové ceny a váhy cena a váha i-tého předmětu. Tím se snižuje počet operací sčítání a odčítání, které se průměrně provedou 2krát, narozdíl od n/2krát u počítání sumy až po vytvoření masky.

Po vypočtení hodnot předmětů v batohu každé kombinace se kontroluje, zda daná kombinace nepřesahuje váhu batohu, a pokud jej nepřesahuje, aktualizuje se nejlepší dosažená hodnota a maska, která k ní patří.

Výpis kódu 5.2 metoda solve řešící problém batohu

```
void KnapsackSolver::solve() {
    if (solved) return;
    std::vector<bool> itemMask(items.size(), false);
    std::pair<unsigned long, unsigned long> itemSum (0, 0);
    while (true) {
        unsigned int i = 0;
        while (i < items.size()) {
            if (itemMask[i]) {
                itemMask[i] = false;
                itemSum.first -= items[i].first;
                itemSum.second -= items[i].second;
            } else {
                itemMask[i] = true;
                itemSum.first += items[i].first;
                itemSum.second += items[i].second;
                break;
            }
            ++i;
        }
        if (i >= items.size()) break;
        if (itemSum.second <= maxWeight && itemSum.first > bestValue) {
            bestValue = itemSum.first;
            std::copy(itemMask.begin(), itemMask.end(), bestItemMask.
                begin());
        }
    }

    solved = true;
}
```

Kapitola 6

Implementace vícevláknových aplikací

Tato kapitola popisuje návrh paralelizace sekvenční aplikace a implementaci pomocí jednotlivých modelů paralelizace v C++.

6.1 implementace společné části vícevláknových aplikací

Abstraktní třída *MultithreadedKnapsackSolver*, popsaná ve výpisu kódu 6.1, definuje atributy a implementuje metody používané všemi 3 vícevláknovými aplikacemi. Stejně jako u třídy *KnapsackSolver* ze sekvenční aplikace, definuje tato třída atributy k uložení dané instance problému batohu a jejímu řešení, metody k načtení instance ze standartního vstupu a vypsání výsledku na standartní výstup. Metoda *solve* není implementována, jelikož implementace této funkce závisí na použitém modelu paralelizace.

■ **Výpis kódu 6.1** Abstraktní třída MultithreadedKnapsackSolver

```
class MultithreadedKnapsackSolver {
protected:
    std::vector<std::pair<unsigned int, unsigned int>> items;
    std::vector<bool> fixedItemMask;
    unsigned long maxWeight;
    unsigned long bestValue;
    std::vector<bool> bestItemMask;
    unsigned int threadsNum;
    bool combinationsCompleted;
    bool solved;
    void processFixedMask(const std::vector<bool> & fixedMask,
                          std::pair<unsigned long, unsigned long> &
                              itemSum,
                          unsigned long & localBestValue,
                          std::vector<bool> & bestMask);
    unsigned long knapsackSolve(std::vector<bool> & fixedItemMask, std::
        vector<bool> & bestItemMask);
    void setFixedMask();
    bool getFixedMask(std::vector<bool> & fixedMaskCopy);
    void setBestResult(std::vector<bool> & itemMask, unsigned long
        besValue);
public:
    MultithreadedKnapsackSolver();
    MultithreadedKnapsackSolver(std::vector<std::pair<unsigned int,
        unsigned int>> & items, unsigned long maxWeight);
    void setThreadsNum(unsigned int threadsNum);
    bool loadFromStdIn();
    unsigned long getResult(std::vector<bool> & bestItemMask);
    void printResult();
    virtual void solve() = 0;
};
```

Atribut *fixedItemMask* reprezentuje fixní část masky a je sdíleným datovým zdrojem pro všechny vlákna, skrze který získávají podmnožiny kombinací předmětů které mají řešit. Výpis kódu 6.2 ukazuje metodu *getFixedMask* skrze kterou vlákna získávají jednotlivé variace masky. Vzhledem k tomu že maska je sdílená, výpočet následující masky bude prováděn v kritické sekci. Z tohoto důvodu není současně s maskou počítána i počáteční cena a váha předmětů aby došlo k omezení času stráveném v této kritické sekci.

Výpis kódu 6.2 metoda getFixedMask

```
bool MultithreadedKnapsackSolver::getFixedMask(std::vector<bool> &
    fixedMaskCopy) {
    if (combinationsCompleted) return false;
    fixedMaskCopy = fixedItemMask;
    std::vector<bool>::iterator it = fixedItemMask.begin();
    while (it != fixedItemMask.end()) {
        if (*it) {
            *it = false;
        } else {
            *it = true;
            break;
        }
        ++it;
    }
    if (it == fixedItemMask.end()) {
        combinationsCompleted = true;
    }
    return true;
}
```

Pomocí metody setBestResult, zobrazené ve výpisu kódu 6.3, porovnávají vlákna své dílčí výsledky s nejlepší dosaženou hodnotou a pokud je větší aktualizují tuto hodnotu. Vzhledem k tomu že dochází k modifikaci sdílených dat mezi vlákny, vlákna by k této metodě přistupovat jednotlivě nebo hrozí riziko race conditions.

Výpis kódu 6.3 Metoda setBestResult

```
void MultithreadedKnapsackSolver::setBestResult(std::vector<bool> &
    itemMask, unsigned long value) {
    if (value > bestValue) {
        bestValue = value;
        bestItemMask.swap(itemMask);
    }
}
```

výpis kódu 6.4 zobrazuje metodu *setFixedMask* sloužící k vytvoření fixní masky předmětů na základě známého počtu vláken která budou použita k výpočtu a počtu předmětů. Velikost je vypočítaná způsobem popsaným v sekci 4.4.

Výpis kódu 6.4 metoda setFixedMask

```
void MultithreadedKnapsackSolver::setFixedMask() {
    unsigned int size = ((unsigned int) (ceil(log2(threadsNum))));
    size += (unsigned int) (log2(items.size() - size));
    size = size > items.size() ? items.size() : size;
    fixedItemMask = std::vector<bool> (size, false);

}
```

Řešení problému batohu pro kombinace s danou fixní částí masky je implementováno metodou *knapsackSolve* zobrazené ve výpisu kódu 6.5, která vychází z řešení problému batohu použitém v sekvenční aplikaci. Této metodě je parametrem předána fixní maska na základě které se vypočítá

výhozí cena a váha předmětů v batohu metodou *processFixedMask*. Následně jsou zkoušeny pouze kombinace $n - d$ posledních předmětů.

Výpis kódu 6.5 Metoda knapsackSolve

```
unsigned long MultithreadedKnapsackSolver::knapsackSolve(std::vector<
    bool> & fixedMask, std::vector<bool> & bestMask) {
    std::vector<bool> itemMask(items.size(), false);
    unsigned long localBestValue = 0;
    std::pair<unsigned long, unsigned long> itemSum(0, 0);
    processFixedMask(fixedMask, itemSum, localBestValue, bestMask);

    while (true) {
        unsigned int i = fixedMask.size();
        while (i < items.size()) {
            if (itemMask[i]) {
                itemMask[i] = false;
                itemSum.first -= items[i].first;
                itemSum.second -= items[i].second;
            } else {
                itemMask[i] = true;
                itemSum.first += items[i].first;
                itemSum.second += items[i].second;
                break;
            }
            ++i;
        }
        if (i == items.size()) break;

        if (itemSum.second <= maxWeight && itemSum.first >
            localBestValue) {
            localBestValue = itemSum.first;
            std::copy(itemMask.begin() + fixedMask.size(), itemMask.end
                (), bestMask.begin() + fixedMask.size());
        }
    }
}
```

6.2 Implementace POSIXovými vlákny

Vícevláknové řešení modelem POSIX vláken zajišťuje třída *POSIXKnapsackSolver*, zobrazená ve výpisu kódu 6.6. Kromě implementace metody *solve*, tato třída obsahuje podpůrné privátní metody a dva atributy maskMutex a bestValueMutex sloužící k identifikaci mutexů, sloužících k synchronizaci přístupu vláken ke sdíleným datům.

Výpis kódu 6.6 Třída POSIXKnapsackSolver

```
class POSIXKnapsackSolver : public MultithreadedKnapsackSolver {
private:
    pthread_mutex_t maskMutex;
    pthread_mutex_t bestValueMutex;
    void thread();
    static void * threadHelper(void * solverInstance);
public:
    POSIXKnapsackSolver();
    POSIXKnapsackSolver(std::vector<std::pair<unsigned int, unsigned int
        >> & items, unsigned long maxWeight);
    void solve() override;
};
```

Metoda *solve*, popsaná ve výpisu kódu 6.7, samotná neřeší problém batohu, ale pouze se stará o správu vláken a mutexů. Před vytvořením vláken se nejdříve inicializují mutexy funkcí *pthread_mutex_init* a vytvoří se fixní část masky, jelikož počet předmětů a vláken se již měnit nebude. V prvním cyklu se vytvoří vlákna dle zadaného počtu funkcí *pthread_create*. Následně se na všechna vlákna v cyklu zavolá funkce *pthread_join*, která čeká dokud dané vlákno nedokončí svou práci. Po spojení všech vytvořených vláken do jednoho lze bezpečně zničit mutexy *pthread_mutex_destroy*.

■ **Výpis kódu 6.7** Metoda solve implementována POSIXovými vlákny

```
void POSIXKnapsackSolver::solve() {

    if (solved) return;

    if (pthread_mutex_init(&maskMutex, NULL) != 0) {
        printf("Error:␣unable␣to␣create␣mutex\n");
        return;
    }
    if (pthread_mutex_init(&bestValueMutex, NULL) != 0) {
        printf("Error:␣unable␣to␣create␣mutex\n");
        return;
    }

    setFixedMask();

    std::vector<pthread_t> threads(threadsNum);
    for (unsigned int i = 0; i < threadsNum; ++i) {
        if (pthread_create(&threads[i], NULL, &POSIXKnapsackSolver::
            threadHelper, this) != 0) {
            printf("Error:␣unable␣to␣create␣threads");
            return;
        }
    }
    for (unsigned int i = 0; i < threadsNum; ++i) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&maskMutex);
    pthread_mutex_destroy(&bestValueMutex);

    solved = true;
}
```

Funkce *pthread_create* předává vláknu funkci, kterou má vykonat. Tato funkce musí mít návratovou hodnotu a jeden parametr, oba ukazatel typu void. V tomto případě to znamená že nelze přímo předat k vykonávání metodu *thread*, která implementuje chování vlákna. Místo toho je při vytváření vlákna předána k vykonání pomocná statická metoda *threadHelper*, popsaná ve výpisu kódu 6.8. Tato statická metoda přijme jako parametr ukazatel na objekt, který vlákna vytváří, přetypuje ho a zavolá pomocí něj metodu *thread*.

■ **Výpis kódu 6.8** statická metoda threadHelper

```
void * POSIXKnapsackSolver::threadHelper(void * solverInstance) {
    POSIXKnapsackSolver * solver = (POSIXKnapsackSolver * )
        solverInstance;
    solver->thread();
    return nullptr;
}
```

Výpis kódu 6.9 zobrazuje implementaci metody *thread*, kterou provádí jednotlivá vlákna, implementující cyklus popsaný v 4.4.2. Přístup ke sdíleným datům je řízen získáváním zámků. před sekcí kódu, kdy dochazí k získání kombinace masky nebo k porovnání a aktualizaci nejlepších dosažených výsledků, se vlákno pokusí získat odpovídající mutex funkcí *pthread_mutex_lock*. Po dokončení operací nad sdílenými daty se mutex uvolní funkcí *pthread_mutex_unlock*. Pokud se

vlákno pokusí získat mutex které již vlastní jiné vlákno, je blokováno do doby dokud není mutex uvolněn, čímž je zaručeno že nedochází k race conditions.

■ **Výpis kódu 6.9** Metoda thread implementovaná POSIXovými vlákny

```
void POSIXKnapsackSolver::thread() {
    std::vector<bool> fixedMaskCopy(fixedItemMask.size(), false);
    std::vector<bool> resultBestMask(items.size(), false);
    while (true) {

        pthread_mutex_lock(&maskMutex);
        if (!getFixedMask(fixedMaskCopy)) {
            pthread_mutex_unlock(&maskMutex);
            break;
        }
        pthread_mutex_unlock(&maskMutex);

        unsigned long resultBestValue = knapsackSolve(fixedMaskCopy,
           resultBestMask);

        pthread_mutex_lock(&bestValueMutex);
        setBestResult(resultBestMask, resultBestValue);
        pthread_mutex_unlock(&bestValueMutex);
    }
}
```

6.3 Implementace C++ vlákny

Vícevláknové řešení modelem C++ vláken implementuje třída *CPPKnapsackSolver*, zobrazená ve výpisu kódu 6.10. Tato třída poskytuje implementaci metody *solve* a podpůrnou metodu *thread*, kterou vykonávájí jednotlivá vlákna. Atributy *bestMutex* a *bestValueMutex* jsou mutexy použité k řízení přístupu ke sdíleným datovým prostředkům.

■ **Výpis kódu 6.10** Třída CPPKnapsackSolver

```
class CPPKnapsackSolver : public MultithreadedKnapsackSolver {
private:
    std::mutex maskMutex;
    std::mutex bestValueMutex;
    void thread();
public:
    CPPKnapsackSolver();
    CPPKnapsackSolver(std::vector<std::pair<unsigned int, unsigned int>>
         & items, unsigned long maxWeight);
    void solve() override;
};
```

Metoda *solve*, zobrazená ve výpisu kódu 6.11, samotná neřeší problém batohu, ale pouze vytváří vlákna která ho řeší. Před vytvořením vláken nastaví fixní masku, jelikož v tuto chvíly jsou počty vláken a počet předmětu již daný. Následně se v cyklu vytvoří specifikovaný počet vláken, reprezentované objekty *std::thread*. Jako parametr konstruktoru vlákna je předán lambda výraz, který pouze zavolá metodu *thread* v kontextu této instance třídy *CPPKnapsackSolver*. V dalším cyklu se volá pro každou instanci třídy *std::thread* metoda *join*, která blokuje hlavní vlákno, dokud se vlákno, reprezentované danou instancí, nedokončí.

■ **Výpis kódu 6.11** Metoda solve implementovaná pomocí C++ vláken

```
void CPPKnapsackSolver::solve() {
    if (solved) return;

    setFixedMask();

    std::vector<std::thread> threads;
    for (unsigned int i = 0; i < threadsNum; ++i) {
        threads.push_back(std::thread([this] { thread(); }));
    }

    for (auto & thread : threads) {
        thread.join();
    }
    solved = true;
}
```

Výpis kódu 6.12 zobrazuje implementaci metody *thread*, kterou provádí jednotlivá vlákna implementující cyklus popsaný v 4.4.2. Přístup ke sdíleným datům je řízen získáváním zámků. Před sekcí kódu, kdy dochazí k získání kombinace masky nebo k porovnání a aktualizaci nejlepších dosažených výsledků, se vlákno pokusí získat odpovídající mutex zavoláním metody *lock*. Po dokončení operací nad sdílenými daty se mutex uvolní metodou *unlock*. Pokud se vlákno pokusí získat mutex které již vlastní jiné vlákno, je blokováno do doby dokud není mutex uvolněn, čímž je zaručeno že nedochází k race conditions.

■ **Výpis kódu 6.12** Metoda thread implementovaná pomocí C++ vláken

```
void CPPKnapsackSolver::thread() {
    std::vector<bool> fixedMaskCopy;
    std::vector<bool> resultBestMask(items.size(), false);

    while (true) {
        maskMutex.lock();
        if (!getFixedMask(fixedMaskCopy)) {
            maskMutex.unlock();
            break;
        }
        maskMutex.unlock();

        unsigned long resultBestValue = knapsackSolve(fixedMaskCopy,
            resultBestMask);

        bestValueMutex.lock();
        setBestResult(resultBestMask, resultBestValue);
        bestValueMutex.unlock();
    }
}
```

6.4 Implementace pomocí OpenMP

Vícevláknové řešení modelem OpenMP implementuje třída *OMPKnapsackSolver*, zobrazená ve výpisu kódu 6.14. Tato třída pouze poskytuje implementaci metody *solve*.

Výpis kódu 6.13 Třída OMPKnapsackSolver

```
class OMPKnapsackSolver : public MultithreadedKnapsackSolver {
public:
    OMPKnapsackSolver();
    OMPKnapsackSolver(std::vector<std::pair<unsigned int, unsigned int>>
        & items, unsigned long maxWeight);
    void solve() override;
};
```

Metoda *solve*, zobrazená ve výpisu kódu 6.14, nejdřívé vytvoří fixní masku a poté následuje cyklus, odpovídající cyklu vlákna popsaném v 4.4.2. Tento cyklus je označen direktivem pro kompilátor *#pragma omp parallel*, který specifikuje že daný blok kódu bude prováděn paralelně počtem vláken, specifikovaným v klauzuli *num_threads*. V rámci této direktivy jsou specifikovány privátní proměnné které zajistí že každé vlákno bude mít svou kopii. Proměnné označené klauzulí *firstprivate* budou mít hodnotu stejnou jako mimo paralelní region.

Přístup ke sdíleným datům je synchronizován pomocí direktivy *pragma omp critical*, zaručující že v daném bloku kódu se v jednu chvíli může nacházet pouze jedno vlákno a nebude docházet k race condition.

Výpis kódu 6.14 Metoda solve implementována pomocí OpenMP

```
void OMPKnapsackSolver::solve() {
    if (solved) return;
    setFixedMask();
    std::vector<bool> fixedMaskCopy;
    std::vector<bool> resultBestMask(items.size(), false);
    bool finished = false;

    #pragma omp parallel private(fixedMaskCopy) firstprivate(
       resultBestMask, finished) num_threads(threadsNum)
    {
        while (true) {
            #pragma omp critical(FIXED_MASK_SECTION)
            {
                finished = !getFixedMask(fixedMaskCopy));
            }
            if (finished) break;

            unsigned long resultBestValue = knapsackSolve(fixedMaskCopy,
                 resultBestMask);

            #pragma omp critical(RESULT_SECTION)
            {
                setBestResult(resultBestMask, resultBestValue);
            }
        }
    }
    solved = true;
}
```

Kapitola 7
Testování aplikací

7.1 Korektnost řešení

U všech 4 aplikací byla otestována korektnost řešení problému batohu. Vzhledem k exponenciální časové složitosti řešení byly testovány instance problému batohu do 30 předmětů. Testovací data byla generována náhodně. K získání referenčních výsledků byl použit Python program rešící problém batohu dynamickým programováním (kód dostupný z [6]). Vzhledem k možné existenci více kombinací předmětů které dosahují celkové nejvyšší ceny byly, porovnávány byly pouze nejvyšší dosažené ceny předmětů v batohu. Korektnost kombinace předmětů byla testována pouze, zda-li celková cena odpovídá řešení a splňuje váhový limit.

U vícevláknových aplikací byly zadávány i zvýšenné počty vláken (až 100) z důvodu větší šance na projevení případných race conditions nebo deadlocků a ověření správného způsobu rozdělování podproblémů mezi vlákna.

7.2 Paralelizace

Správné vytváření vláken vícevláknovými aplikacemi bylo ověřeno Linuxovým příkazem *ps*. Počet vláken procesu je ve výstupu označen jako NLWP (number of light weight processes)

Pro ověření že vícevláknové aplikace využívájí správně jádra CPU byl použit nástroj *htop*, pro monitorování systému v operačním systému Linux. Pomocí tohoto nástroje lze získat informace o využití CPU jednotlivými procesy.

Kapitola 8

Používání Aplikací

Všechny aplikace příjímají instanci problému batohu k řešení skrze standardní vstup. Na prvním řádku se specifikuje počet předmětů a maximální kapacita batohu. Následuje počet řádků odpovídající specifikovaném počtu předmětů, každý obsahuje dvojice čísel specifikující cenu a váhu daného předmětu. Aplikace přijímá instance ze standardního vstupu dokud jsou dostupná data. Výsledek je zobrazen na standardní výstup ve formátu nejvyšší možné cenu předmětů v batohu a binární masky reprezentující odpovídající kombinaci předmětů v batohu.

U vícevláknových aplikací lze specifikovat počet použitých vláken k výpočtu parametrem při spuštění. Pokud tento parametr není předán, je použita defaultní hodnota.

Obrázek 8.1 zobrazuje použití vícevláknové aplikace pro vyřešení jedné instance s 5 předměty 6 vlákny.

■ **Obrázek 8.1** Ukázka použití vícevláknové aplikace

```
./POSIXKnapsack 6
5 30
14 8
11 12
24 13
8 7
9 4
------------------------------
47
10101
```

Kapitola 9

Náročnost implemetace vícevláknových aplikací

Tato kapitola porovnává náročnost implementace vícevláknových aplikací. V rámci implemetace vícevláknových aplikací byl zaznamenáván čas využitý k jejich implementaci a odladění, zobrazený v tabulce 9.1.

Tabulka 9.1 Tabulka časové náročnosti implementace vícevláknových aplikací

časová náročnost implementace vícevláknových aplikací			
model	implementace (min)	odladění (min)	celkem (min)
POSIX vlákna	75	30	300
C++ vlákna	60	15	270
OpenMP	40	20	255
Společná část	135	60	-

Z tabulky lze vidět že nejvíce času zabrala implementace a odladění společné části, uzpůsobující algoritmus pro řešení problému bathu hrubou silou k paralelizaci. Důsledkem tohoto je, že celková doba implementace jednotlivých vícevláknových aplikací se příliš neliší. Pokud se ale díváme čistě na rozdíly mezi dobou potřebnou na paralelizaci kódu, který je tomu již uzpůsoben, rozdíly jsou markantnější.

Nejnáročnějším modelem pro paralelizaci z hlediska času byla POSIX vlákna. Přestože POSIX vlákna a C++ vlákna jsou v rámci této práce používána podobným způsobem z hlediska míry kontroly nad vlákny, C++ vlákna jsou intuitivnější a více uživatelsky přívětivé.

Implementace pomocí OpenMP byla nejméně časově náročná z důvodu vysoké míry abstrakce, která ušetří spoustu práce s vytvářením a následným slučováním vláken a řízením přístupu ke sdíleným paměťovým prostředkům. Paralelizace kódu skrze direktiva pro překladač, umožňuje bez výrazných zásahů do kódu paralelizovat algoritmus který je pro paralelizaci vhodný. Samotná direktiva navíc zpřehledňují kód, ze kterého je na první pohled vidět které části jsou vykonávány paralelně.

Kapitola 10
Testování zrychlení

Tato kapitola popisuje měření časové náročnosti jednotlivých aplikací a prezentuje výsledky, které jsou vyhodnoceny.

10.1 Metodika měření

Měření času

Čas byl měřen pouze pro proces nalezení řešení problému batohu odpovídající volání metody *solve* jednotlivých tříd a nezahrunuje načtení vstupu a výpis výsledku. K samotnému měření byla využita knihovna *chrono*, která je součástí standardní knihovny jazyka C++.

Kompilace

Ke kompilaci aplikací byl použit kompilátor g++ verze 11.2. s přepínačem optimalizace -O3.

Hardware

Aplikace byly testovány na procesoru *intel core i7-8750H 2.2 GHz* disponující 6 jádry.

Data

K testování byla použita náhodně vygenerovaná data, kde ceny i váhy jednotlivých předmětů byly vybírány nezávisle na sobě z intervalu $[1, 10000]$ a celková kapacita z intervalu $[1, 10000n]$ kde n je počet dostupných předmětů. Vzhledem k povaze použitého algoritmu řešení hrubou silou, nebyly uvažovány sady instancí, mající nějakou míru korelace vah a cen předmětů.

10.2 Výsledky měření

Měření bylo prováděno vždy pro 100 instancí problému batohu. Časy (T) v tabulkách jsou zprůměrovány a uvedeny v milisekundách. Dále tabulky obsahují zrychlení (Z) a efektivitu (E). Tabulka zobrazuje výsledky pouze pro instance problému s 28 až 31 předměty z důvodu exponenciální časové složitosti použitého algoritmu.

Tabulka 10.1 Tabulka výsledků měření sekvenční aplikace

Naměřené hodnoty sekvenční aplikace	
n	T
27	753
28	1495
29	3014
30	6007

Tabulka 10.2 Tabulka výsledků měření vícevláknové aplikace implementované POSIX vlákny

Naměřené hodnoty aplikace implementované POSIX vlákny									
počet vláken	2			4			6		
n	T	Z	E	T	Z	E	T	Z	E
27	421	1,789	0,894	223	3,502	0,876	153	5,158	0,860
28	837	1,786	0,893	431	3,509	0,877	296	5,102	0,850
29	1706	1,767	0,883	855	3,492	0,873	572	5,179	0,863
30	3296	1,806	0,903	1737	3,458	0,865	1167	5,147	0,858

Tabulka 10.3 Tabulka výsledků měření vícevláknové aplikace implementované C++ vlákny

Naměřené hodnoty aplikace implementované C++ vlákny									
počet vláken	2			4			6		
n	T	Z	E	T	Z	E	T	Z	E
27	437	1,723	0,862	221	3,407	0,852	151	4,987	0,831
28	852	1,755	0,877	434	3,445	0,861	297	5,034	0,839
29	1706	1,767	0,883	871	3,460	0,865	591	5,100	0,850
30	3372	1,781	0,891	1734	3,464	0,866	1203	4,993	0,832

Tabulka 10.4 Tabulka výsledků měření vícevláknové aplikace implementované OpenMP

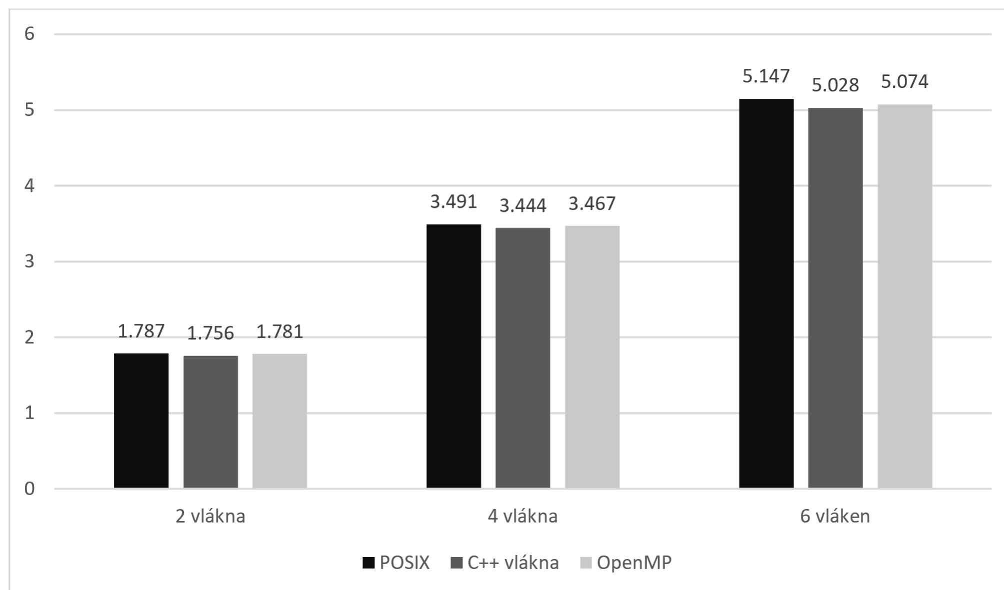
Naměřené hodnoty aplikace implementované OpenMP									
počet vláken	2			4			6		
n	T	Z	E	T	Z	E	T	Z	E
27	425	1,772	0,886	218	3,454	0,864	150	5,020	0,837
28	829	1,803	0,902	429	3,485	0,871	296	5,051	0,842
29	1711	1,762	0,881	867	3,476	0,869	586	5,143	0,857
30	3362	1,787	0,893	1739	3,454	0,864	1182	5,082	0,847

10.3 Zhodnocení výsledků

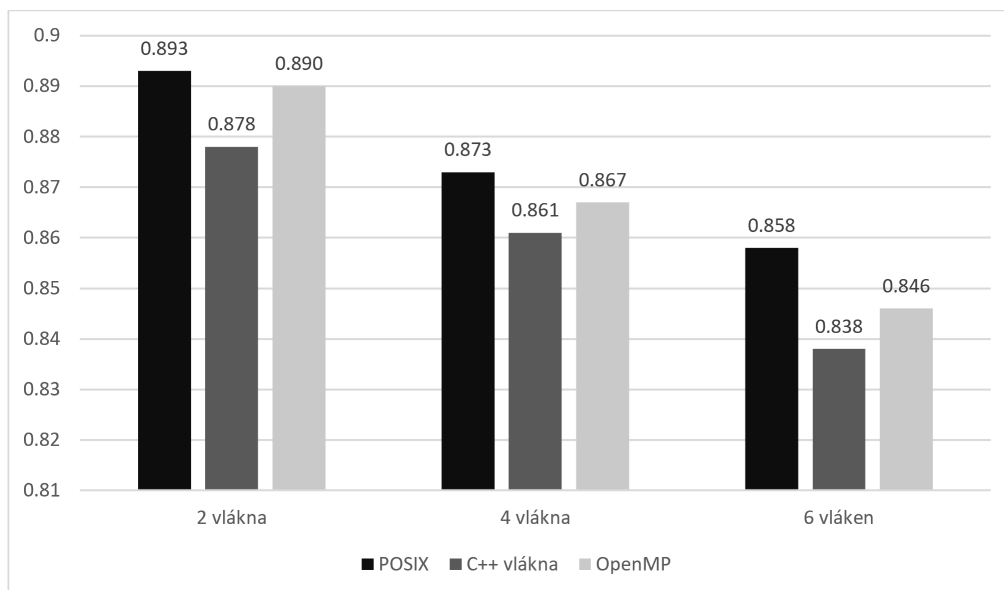
Zrychlení dle očekávání roste s počtem vláken. Z grafu 10.2 zobrazující průměrnou efektivitu všech měřených instancí v závislosti na počtu vláken, ale lze vidět že efektivita s rostoucím počtem vláken klesá, z důvodu zvýšenné systémové režii a zvýšenému poměru sekvenčních částí ku paralelním.

Mezi jednotlivými implementacemi se projevily malé rozdíly v efektivitě a tedy dosaženém zrychlení. Největšího průměrného zrychlení dosahuje aplikace implementovaná POSIX vlákny, následovaná aplikací využívající OpenMP. Nejnižšího průměrného zrychlení dosahovala aplikace implementovaná C++ vlákny.

Obrázek 10.1 Průměrná efektivita v závislosti na počtu vláken



Obrázek 10.2 Průměrné zrychlení v závislosti na počtu vláken



Závěr

Tato práce se zaměřila na porovnání různých vícevláknových modelů pro paralelní programování v jazyce C++. V rámci práce byla vytvořena sekvenční aplikace řešící problém batohu 0-1 hrubou silou a následně byla paralelizována pomocí 3 různých vícevláknových modelů .

Součástí práce je analýza časové náročnosti na paralelizaci a dosaženého zrychlení na vícejádrovém systému. Celkově lze konstatovat, že pro paralelizaci problému batohu 0-1 hrubou silou je nejvhodnější využití modelu OpenMP. Tento model nabízí jednodušší implementaci se srovnatelým dosaženým zrychlením jako C++ vlákna a POSIX vlákna. Zároveň je ale třeba poznamenat, že správná volba modelu paralelního programování je závislá na konkrétním řešeném problému a platformě pro nasazení.

Možným rozšířením této práce by mohla být použití většího vzorku programátorů pro analýzu časové náročnosti na paralelizace použitím jednotlivých modelů.

Literatura

[1] PACHECO, Peter S. *An introduction to parallel programming.* Amsterdam: Morgan Kaufmann, c2011. ISBN 978-0-12-374260-5

[2] MARTELLO, Silvano; TOTH, Paolo. *Knapsack Problems: Algorithms and Computer Implementations.* New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN 0-471-92420-2

[3] BARNEY, Blaise. *POSIX Threads Programming.* Lawrence Livermore National Laboratory, [online]. c2010 [cit. 2023-05-08]. Dostupné z: `https://hpc-tutorials.llnl.gov/posix`

[4] *Concurrency support library.* In: *cppreference.com* [online]. [Cit. 2023-05-08]. Dostupné z: `https://en.cppreference.com/w/cpp/thread`

[5] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.2.* [online]. c2021 [cit. 2023-05-08]. Dostupné z: `https://www.openmp.org/specification`

[6] MAO, Lei *Solving 0-1 Knapsack Problems* In: leimao.github.io [online]. 13.02.2023 [cit. 2023-05-08]. `https://leimao.github.io/blog/Solving-Knapsack-Problems/`

Příloha A

Obsah přiloženého média

```
readme.txt ........ stručný popis obsahu média
├── exe ........ adresář se spustitelnými soubory aplikací a testovacích programů
├── obj ........ adresář s obsahující zkompilované objektové soubory
├── src ........ zdrojové kódy implementace
├── tests ........ složka se soubory obsahující testvací data
├── thesis
│   ├── thesis.pdf ........ text práce ve formátu PDF
│   └── src ........ zdrojová forma práce ve formátu LaTeX
└── Makefile ........ soubor s instrukcemi pro kompilaci a linkování
```