Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Paralelizace v prostředí .NET

Vít Jašek

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Vít Jašek

Informatika

Název práce

Paralelizace v prostředí .NET

Název anglicky

Parallelization in .NET framework

Cíle práce

Práce je zaměřena na problematiku paralelního programování v jazyce C#. Cílem práce je na ukázkových aplikacích demonstrovat možnosti využití Task Parallel Library a dále ukázat pro které problémové oblasti využití paralelizace přináší výhody a pro které nikoliv.

Metodika

Práce sestává ze dvou hlavních částí – přehledu teoretických východisek a praktické části.

Metodika zpracování teoretické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků bude popsána problematika vývoje aplikací v jazyce C# s využitím paralelizace.

V praktické části práce budou s využitím jazyka C# implementovány ukázkové aplikace řešící vybrané problémy, na kterých budou demonstrovány možnosti využití Task Parallel Library v prostředí .NET. U jednotlivých příkladů bude provedeno zhodnocení výsledku paralelizace řešení daného problému a na základě těchto poznatků budou formulovány závěry bakalářské práce.

Doporučený rozsah práce

35-40 stran

Klíčová slova

C#, .NET, paralelizace, task, thread, TPL

Doporučené zdroje informací

FREEMAN, Adam. Pro .NET 4 Parallel Programming in C#. ISBN 978-1-4302-2968-1.

Microsoft Developer Network [online]. Dostupné také z:
https://msdn.microsoft.com/cs-cz/library/dd460717(v=vs.110).aspx

NAKOV, Svetlin & kol. Fundamentals of Computer Programming with C#: The Bulgarian C# Book [online]. Dostupné z:
http://www.introprogramming.info/wp-content/uploads/2013/07/Books/CSharpEn/Fundamentals-of-Computer-Programming-with-CSharp-Nakov-eBook-v2013.pdf

---

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 11. 1. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 14. 03. 2018

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Paralelizace v prostředí .NET" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne _____7.3.2018_____

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu Ing. Jiřímu Brožkovi, Ph.D. za cenné rady a konzultace, své ženě za podporu a rodičům za trpělivost.

Paralelizace v prostředí .NET

Abstrakt

Práce je zaměřena ukázky aplikací a možnosti jejich paralelizace v jazyce C# v .NET Frameworku. Při testování jsou srovnávány časy vykonání synchronního(sériového) přístupu s paralelním. V aplikacích jsou ukázány přístupy a myšlenkové modely, jak postupovat při paralelizaci. Dále je demonstrováno, kdy paralelismus přináší užitek a kdy ne. Všechny programy jsou pak schopné hromadného testu, při kterém se daná operace provede více násobně.

Klíčová slova: C#, .NET, paralelizace, task, thread, TPL

Parallelization in .NET framework

Abstract

This bachelor thesis is aimed at application samples and possibilities of their parallelization in C# language in .NET Framework. The time of execution of the synchronic (serial) approach is compared with the time of the parallel approach during testing. The approaches and thought models of the process during parallelization are shown in the applications. Furthermore there are demonstrated the situations where we can benefit from parallelism and when there is no need for it. All programmes are able of a collective test, during which the given operation performs multiple times.

Keywords: C#, .NET, parallelization, task, thread, TPL

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Když v roce 2015 byly oznámeny první 2 jádrové procesory pro PC platformu vedly se veliké diskuze nad tím, jestli koupit výkonné jedno jádro nebo jestli naopak koupit dvou jádro se dvěma slabšími jádry. Co vlastně je budoucnost? Obrovitánský frekvence a výkon na jedno jádro nebo menší frekvence, ale více jader? Dnes je již situace jasná. Zvítězilo více jader, částečně i z důvodů, že frekvence se nedá zvyšovat do nekonečna. Pro programátory to znamená „nový trend“ v programování. V uvozovkách jsem to napsal schválně, protože programovací jazyky uměly vlákna už před více jádrovými procesory.

V dnešní době telefony mají běžně 8 jádrový procesor a v počítačích se z 4 jádra stává nutný minimum. Paralelní programování je tedy důležitý um, který by dnes měl každý programátor zvládat. Tento styl vytváření programů s sebou přináší krom příjemného nárůstu výkonu také spoustu úskalí. Vláknový kód se obvykle velmi špatně krokuje a tak jeho ladění není nic jednoduchého a příjemného. Dále, může se stát, že i při vší snaze o paralelizaci, program stejně nezrychlí ba naopak. V některých případech může být paralelizace tak neefektivní, že je lepší od ní odstoupit.

Ve své práci ukážu případy, kdy je paralelizace opravdu velmi efektivní a kód paralelně napsaný dokáže ušetřit velmi mnoho času. Ukážu ale i případy, kdy je paralelizace neefektivní a tím pádem zbytečná.

2 Cíl práce a metodika

2.1 Cíl práce

Práce je zaměřena na problematiku paralelního programování v jazyce C#. Cílem práce je na ukázkových aplikacích demonstrovat možnosti využití Task Parallel Library a dále ukázat pro které problémové oblasti využití paralelizace přináší výhody a pro které nikoliv.

2.2 Metodika

Práce sestává ze dvou hlavních částí – přehledu teoretických východisek a praktické části. Metodika zpracování teoretické části je založena na studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků bude popsána problematika vývoje aplikací v jazyce C# s využitím paralelizace.

V praktické části práce budou s využitím jazyka C# implementovány ukázkové aplikace řešící vybrané problémy, na kterých budou demonstrovány možnosti využití Task Parallel Library v prostředí .NET. U jednotlivých příkladů bude provedeno zhodnocení výsledku paralelizace řešení daného problému a na základě těchto poznatků budou formulovány závěry bakalářské práce.

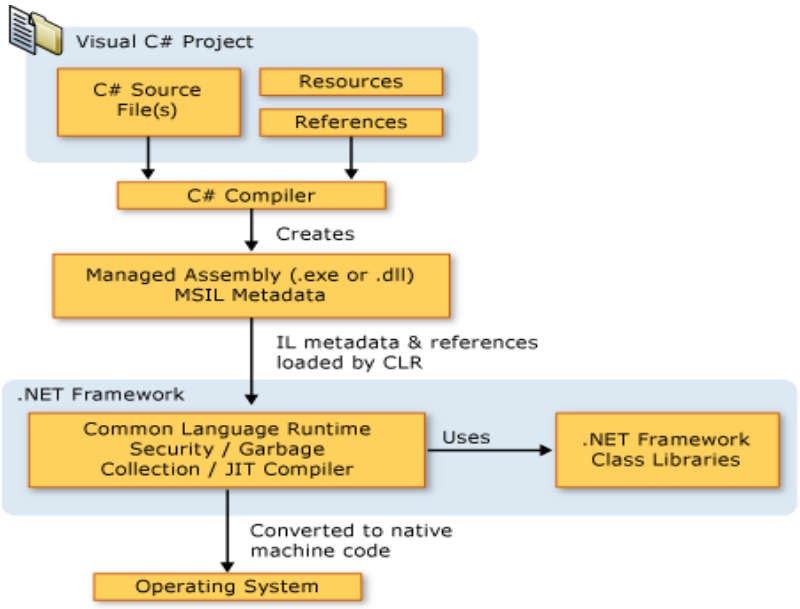
3 Teoretická východiska

3.1 Jazyk C# a prostředí .NET

Jazyk C# je moderní, objektově orientovaný, typově bezpečný, vysoko-úrovňový programovací jazyk, který vznikl společně s .NET platformou v roce 2000 ve společnosti Microsoft. V C# je možné naprogramovat velkou škálu aplikací od jednoduchého webu, po počítačové hry. Syntaxe jazyka je odvozená z jazyka C/C++ a jeho objektově orientovaný přístup je zase velmi podobný jazyku Java. C# je interpretovaný jazyk, to znamená, že se nejdříve přeloží do mezi jazyka a až poté se překládá do strojového kódu. Zdrojový kód programu napsaného v tomto jazyce se přeloží, ovšem nikoli do strojového kódu počítače, ale do universálního pomocného jazyka označovaného Microsoft Intermediate Language. Tento pomocný jazyk se dále překládá do strojového kódu.

V praxi to znamená že na cílovém počítači musíme mít nainstalovaný .NET Framework, který v sobě obsahuje jak interpreter C#[1], tak i JIT překladač do strojového kódu, ale další nezbytné součásti pro běh programů vytvořených v C#. Tato vlastnost, stejně tak jako u jazyku Java, nám zaručuje multiplatformnost jazyka a prostředí .NET. Existují totiž verze .NET framework jak pro Linux tak i pro MacOs, což je veliká motivace pro programování právě v C# .NET. V poslední řadě bych zmínil, že existuje prostředí Xamarin, které je postavené na .NET a C# a toto prostředí nám umožňuje programovat pro mobilní telefony na platformách Microsoft, Android a iOS.

[1] NAKOV & CO., 2013, Svetlin. *FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#*. s. 18.



Obrázek 1 Překlad[2]

3.1.1 Objektové programování v C# (OOP)

Objektově orientované programování je filozofie a způsob myšlení při vytváření programů. Při programování OOP se klade důraz hlavně na znovu-použitelnost kódu a zapouzdření. Mezi základní pilíře OOP také patří dědičnost a polymorfizmus. Objekty programu představují objekty z reálného světa, kde každý objekt obsahuje atributy a metody. [3]

3.1.1.1 Atributy

Atributy jsou lidsky řečeno vlastnosti objektu. Jsou to data, která objekt uchovává po celou dobu svojí instance. Z pohledu programu to není nic jiného než proměnné. [4]

3.1.1.2 Metody

Jestliže atributy jsou proměnné objektu, tak metody jsou jeho funkce. Metody je také vhodné pojmenovat tak, aby při práci s objektem dávaly smysl a hezky se v kódu četly. Například: soubor.vymaz(), zarovka.rozsvit() atd. [5]

---

[2] MSDN. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.
[3] Itnetwork. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani.
[4] Tamtéž
[5] Tamtéž

3.1.1.3 Znovu-použitelnost

Jedná se o styl psaní kódu takovým způsobem, aby programátor, který po nás kód převezme, nemusel zkoumat, co daný objekt dělá, ale jen ho implementoval. V praxi totiž není vždy čas, na to psát celý projekt sám a tak se využití knihoven jeví jako dobrá úspora času. [6]

3.1.1.4 Zapouzdření

Umožňuje pomocí modifikátorů přístupnosti (private, protected, public), ukrýt atributy a metody tak, aby zůstaly přístupné pouze pro třídu zevnitř anebo pro z ní odvozené třídy. K takové třídě potom existuje rozhraní, přes které ji využíváme a předáváme instrukce.[7]

3.1.1.5 Dědičnost

Představme si, že existuje třída předek a k ní třída potomek. Třída potomek má potom všechny vlastnosti třídy předek, a ještě k nim nějaký svoje navíc. Dědičnost může být i vícenásobná, ale zde si musíme dát pozor. C# umí dědit pouze jednu třídu, proto vícenásobnou dědičnost realizuje skrze rozhraní. Dědičnost dále zvyšuje znovu-použitelnost kódu a čitelnost kódu. [8]

3.1.1.6 Polymorfizmus

Znamená také mnohoznačnost, neboli jedna metoda může být volána s různými parametry. Nebo také můžeme v třídě potomek přetížit metodu z třídy předek. Například: Každá třída v C# je vlastně potomkem třídy Object. Třída Object implementuje metodu ToString(). Pokud bychom chtěli v naší třídě tuto metodu, musíme napsat do hlavičky před návratový typ slovíčko override.

3.1.2 Speciální konstrukce jazyka C#

Jazyk C# obsahuje některé speciální konstrukce, které se při paralelizaci hodně využívají a proto je stojí za to je zde vysvětlit.

---

[6] Itnetwork. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani.
[7] Tamtéž
[8] Tamtéž

3.1.2.1 Delegát

Delegát je reference na metodu. Je to jakýsi předpis pro signaturu metody.
```
public delegate double ObsahCtverce(int a);
```
[9]

3.1.2.2 Lambda výraz

Lambda výraz je anonymní funkce, která obsahuje výraz nebo blok kódu který chceme provést a vstupní parametry. Na levé straně lambda výrazu jsou vstupní parametry, napravo pak výraz, který chceme provést. [10]

```
()=>{System.Diagnostics.Debug.WriteLine('zprava');}
```

3.1.2.3 Lock

Klíčové slovo Lock zajišťuje, že když s daným blokem kódu pracuje jedno vlákno, ostatním vláknům je přistup odepřen a musejí čekat dokud pracovní vlákno neukončí svojí práci.

3.2 Myšlenka paralelizmu a metody realizace

Programy psané sekvenčně, tedy běžným způsobem, se také sekvenčně vykonávají. Není na tom nic špatného a většinou nám to tak i stačí. Ale doba výrazně pokročila a výrobci procesorů vyrábějí čipy, které obsahují více než jedno jádro. Nyní můžeme klást otázky. Dokážeme tyto jádra další jádra využít? Není jen další chytrý marketingový tah, jak nám prodat něco, co ve skutečnosti nepotřebujeme/nevyužijeme? Odpovědi zní: Ano dokážeme, ne není. Důvod proč tomu tak je, je velmi jednoduchý. Vykonávání více úloh, což více jádrový procesor umožňuje, šetří náš čas. To, jak využijeme prostředky, který nám více jádrové CPU nabízí, už ovšem záleží na našich programovacích schopnostech. Měli bychom se vždy snažit o zrychlování úměrné počtu jader. Neplatí pravidlo, na více jádrech více rychlosti. Naopak pokud to s paralelizací přeženeme, náš program může i zpomalit. Je tedy třeba dbát na efektivitu. Nemá cenu paralelizovat za každou cenu. Pokud výkon našeho programu s paralelizací nestoupá, tak máme v podstatě 3 možnosti.

---

[9] MSDN. Dostupné z: https://msdn.microsoft.com/en-us/library/ms173171(v=vs.110).aspx
[10] NAKOV & CO., 2013, Svetlin. *FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#*. s. 920

- Provést optimalizace našeho kódu – upravit použité metody, nevytvářet zbytečně objekty navíc apod.
- Chybný návrh programu – počáteční myšlenka nevedla k cíli a je program zapotřebí přepracovat.
- Daný problém nemá řešení pomocí paralelizace – paralelizací se program zpomalí a nebo v horším případě vrací chybné výsledky.

Je potřeba ještě zmínit, že více vláknový program může běžet rychleji i na jedno jádrovém, procesoru, a to díky funkcím jako je Intel Hyperthreading a jim podobné. [11]

V následujících kapitolách ukážu metody, jak paralelizovat některé algoritmy. Existují problémy, které jsou tak jednoduché na paralelizaci, že čtenář bude sám překvapen tím, jak rychle a efektivně lze algoritmus paralelizovat a urychlit. Ovšem existují problémy, které jsou natolik složité, že ve výsledku je lepší od paralelizace odstoupit. Obecně se dá říct, že úlohy na paralelizaci lze rozdělit do následujících dvou skupin. [12]

3.2.1 Paralelizace sekvenčního algoritmu

Už z nadpisu je asi jasné o co nám zde půjde. Vezmeme již hotový sekvenčně napsaný program a zkusíme ho za pomocí paralelních nástrojů jazyka C# paralelizovat. V této metodice je obrovská výhoda a to je, že nemusíme program psát a projektovat celý znova. Bohužel se může stát, že prosedíme několik hodin nad tím, proč nám, z ničeho nic, funkční program přestal fungovat, nebo proč je výstup úplně jiný, než jsme čekali. Další problém, s kterým se zde můžeme setkat, se týká rychlosti. Ač se nám program podaří paralelizovat, tak výsledek je pomalejší než sekvenčně napsaný program. V takovéto situaci jsme na rozcestí. Buď napsat úplně nový paralelní program a nebo se smířit se sekvenčním řešením.

3.2.2 Úplně nový paralelní program

Z předchozího odstavce už víme, že paralelizace původně sekvenčního programu může selhat. Nezbývá nám nic jiného než celý program napsat znova. Algoritmus je potřeba od počátku změnit. Program se většinou zesložití a znečitelní. Ovšem ani úplně nový program

[11] Intel. Dostupné z: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html
[12] FREEMAN, Adam. *Pro .NET 4 Parallel Programming in C#*. s. 3.

není záruka úspěchu. Některé problémy jsou ryze sekvenční a neexistuje způsob, jak je paralelizovat.

3.3 Dekompozice paralelních úloh

V této kapitole se budu věnovat, jak lze přistupovat k řešení problému paralelizace programu. Z předchozích dvou odstavců víme, že lze vytvářet buďto nový program, anebo paralelizovat ten starý, sekvenční. Nyní se budeme věnovat samotné dekompozici. Dekompozici lze rozdělit na dva druhy, a to dekompozici úkolovou a dekompozici datovou.

3.3.1 Úkolová dekompozice

Úkolová dekompozice znamená, že náš program je schopný vykonávat několik úloh nezávisle na sobě. Přičemž ale nejde o velký problém rozdělený na několik menších, ale o několik nezávisle na sobě jdoucích úloh.[13]

3.3.2 Datová dekompozice

Co datová dekompozice vlastně znamená? Vezmu svojí datovou základnu, z které chci něco počítat. Rozdělím ji na příslušné dílky. A tyto dílky poté zpracovávám samostatně. Při paralelním programování je potřeba se postarat o správné zpracování dat. Pokud tak neučiníme, může nastat jev, který se nazývá křížení dat (data race) a náš program nám vrátí úplně jiný výsledek, než očekáváme. Typicky, takováto situace nastává, když dva úkoly sdílejí jedny data, ze kterých čtou a do kterých zapisují. Je zde potřeba dávat pozor na to, které komponenty našeho kódu jsou thread save (bezpečná manipulace s daty při vláknovém zpracování). C# má v sobě nástroje, pomocí kterých lze data race předejít, ale jejich požití obvykle znamená ztrátu výkonu. Proto před jejich požitím doporučuji zvážit jiná řešení. .[14]

3.4 Paralelismus v .NET4.0 vs. starší verze .NET

Podpora paralelizace v .NET frameworku je už pěknou řádku let. Ve verzi 1.1 byla přidána Thread class, která dovolovala paralelizaci pomocí ThreadPool (shromaždiště vláken). Také se zde poprvé objevila komponenta backgroudworker, kterou ukáži ve svých

[13] *Parallel Computing* [online]. Dostupné z: https://msdn.microsoft.com/cs-cz/vstudio/bb964701.
[14] Tamtéž

příkladech. Tvorba paralelních programů pomocí Thread class byla dost složitá a vyžadovala vysokou znalost .NET a programování v něm. Toto všechno se ale ve verzi 4.0 změnilo. Microsoft v této verzi .NET přidal knihovny takzvané „Parallel extensions“. Tyto knihovny jsou tvořeny ze dvou částí.

- Parallel LINQ (PLINQ)
- Task Parallel Library(TPL)

Díky těmto knihovnám se práce s více jádrovými procesory a aplikacemi na ně velice usnadnila a více tak přiblížila i programátorům, co nejsou tak úplně mistři v oboru. Teď se naskýtá otázka, jakou výhodu má TPL oproti threadu? TPL má automatickou správu thread poolu. To znamená, že každému úkolu je automaticky přiřazeno vlákno s ohledem na co největší efektivitu. TPL dále poskytuje jednodušší a pohodlnější kontrolu nad vlákny. Já se ve své práci budu zabývat především (TPL). Pro začátek zde uvedu základní konstrukce, které nám (TPL) nabízí.

3.4.1 Paralelní cyklus

Paralelní cyklus je asi nejjednodušší způsob paralelizace. Framework se postará úplně o všechno a sami na příkladu uvidíte, že to chodí velice pěkně. Paralelní cyklus existuje ve dvou verzích a to jako Parallel.For a poté jako Parallel.ForEach. [15]

3.4.1.1 Parallel.For

```
Parallel.For(0, MatA.GetLength(0), row =>
{
    for (int col = 0; col < MatB.GetLength(1); col++)
    {
        for (int inner = 0; inner < MatA.GetLength(1); inner++)
        {
            Result[row, col] += MatA[row, inner] * MatB[inner, col];
        }
    }
});
```

3.4.1.2 Parallel.ForEach

```
Parallel.ForEach(res, (string str) => {
   DoSomething();
});
```

[15] FREEMAN, Adam. *Pro .NET 4 Parallel Programming in C#*. s. 176.

3.4.2 Tasks

Z překladu už vyplívá že se jedná o jakýsi úkol. Zde už je situace trochu složitější než u paralelních cyklů. Rozlišujeme zde implicitní a explicitní vykonávání.

3.4.2.1 Implicitní vykonávání

Implicitní vykonání Tasku provedeme pomocí metody Parallel.Invoke, která provede námi definované akce s delegáty odkazující na statické metody. Zní to trochu kostrbatě, ale na příkladu uvidíte, že to zase tak hrozné není. Výhodou implicitního vykonávání je, že má velice krátký zápis. Parallel.Invoke(()=>vynasob()); Pokud tedy potřebujeme oddělit nějaký kus kódu od hlavního vlákna, můžeme takto. Důvod proč oddělit kus kódu od hlavního vlákna může být typicky, když nechceme, aby nám okýnko UI zamrzlo.

3.4.2.2 Explicitní vykonávání

Explicitní úkol je instance třídy System.Threading.Tasks. Task a ti z vás, kteří mají zkušenosti s paralelním programováním ve starších verzích .NET než 4.0, zde uvidí jistou podobnost s prací s klasickými threads. Metod pro vytvoření explicitního úkolu je několik:

- Pomocí Action delegáta a metody
  ```
  Task task = new Task(new Action(vynasob));
  ```
- Pomocí anonymního delegáta
  ```
  Task task = new Task(delegate { vynasob(););
  ```
- Pomocí lambda výrazu
  ```
  Task task = new Task(() => { vynasob(); });
  ```
- Pomocí statické metody třídy Task
  ```
  Task.Factory.StartNew(() => { vynasob(); });
  ```

3.4.3 Zrušení Tasku

Někdy je zapotřebí ukončit vykonávání Tasku před jeho dokončením. Díky knihovně TPL už není zapotřebí si psát vlastní ukončovací kód, ale můžeme využít CancellationToken. K němu ještě potřebuje zdroj tokenu CancellationTokenSource. Z něho vytvoříme objekt tokenu a ten předáme jako druhý parametr konstruktoru Tasku. Pak už jen na zdroji tokenu stačí zavolat tokenSource.Cancel(); a vykonávaný task se zruší. [16]

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;
Task task1 = new Task(new Action(myMethod), token);
tokenSource.Cancel();
```

[16] FREEMAN, Adam. *Pro .NET 4 Parallel Programming in C#*.

3.4.4 Měření výkonu

3.4.4.1 Testování aplikací

Každý kus kódu, který naprogramuje musíme i otestovat. V našem případě, kromě testování funkcionality, půjde i o zcela úplně jiné testování. Nepůjde ani tak o ošetření výjimek a správnosti vstupů, ale půjde o testování rychlosti. Bude nás zajímat, jestli naprogramovaný paralelní algoritmus zrychluje a tím přináší užitek, nebo jestli naopak paralelizmus nic nepřináší a je tím spíše na škodu.

3.4.5 Metodika testování

Každý naprogramovaný program budu testovat na počítači s 8 jádrovým procesorem. Protože nás bude hlavně zajímat ušetřený čas exekuce programu, budu testovat dobu vykonání pomocí třídy Stopwatch. Schopnosti třídy nám bohatě postačí na vyjádření výsledků a závěrů. Dále každý program budu testovat vícenásobně, protože může nastat situace, že systém nebo nějaká služba bude potřebovat procesorový čas a tím nám výsledek může zkreslit. Výsledné časy poté zprůměruji a tím by se měla případná chyba eliminovat.

Programy, u kterých to půjde, budu testovat na různě velkých datech. Tím demonstruji od kdy/od jakého množství dat nám začíná paralelizace přinášet tížené zrychlení.

3.4.6 Počet jader vs. počet tasks

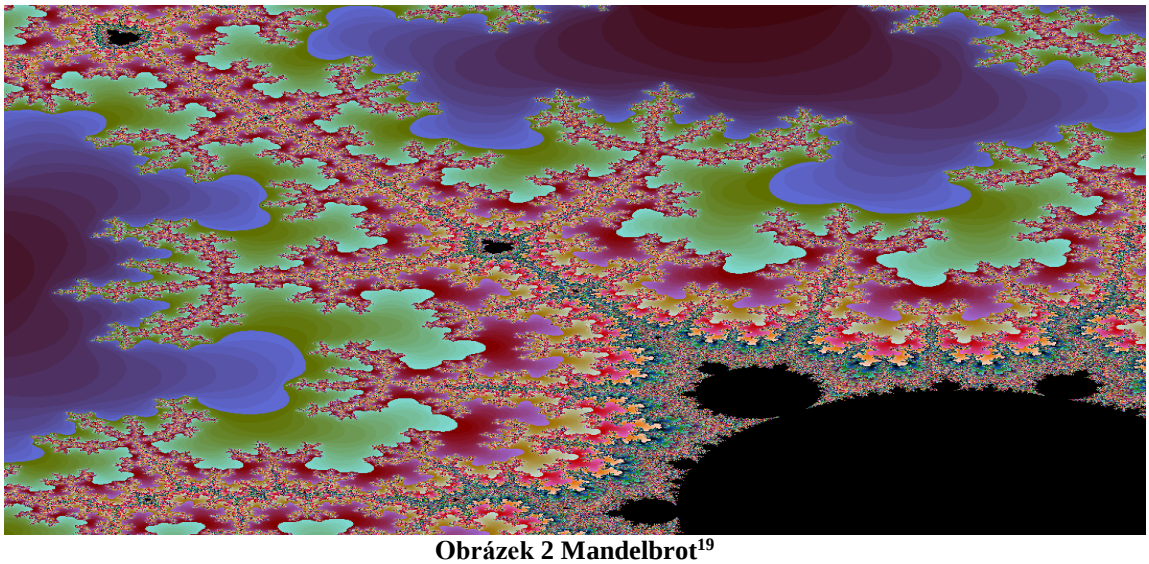
Každého asi napadne, že musí být nejlepší, resp. nejrychlejší, když vytížíme CPU na 100%. Opak však může být pravdou. V tomto odstavci bych se chtěl krátce zmínit o funkci Turbo. Oba přední výrobci PC procesorových čipů (AMD – Turbo Core, Intel – Turbo Boost) mají u svých procesorů funkci, která umí při vytížení zvednout frekvenci právě vytíženého jádra. Čím více jader je vytížených, tím menší je výsledná frekvence vytížených jader (jádra která nic nedělají jsou z úsporných důvodů podtaktovaná).

Úvaha: procesor má 8 jader a úloha je řešitelná do 20 sekund. Nebude výhodnější jí pustit jen na 6 jádrech ale s větší frekvencí? Pravděpodobně bude, proto musíme u úloh řešit i takzvanou škálovatelnost. [17]

[17] FREEMAN, Adam. *Pro .NET 4 Parallel Programming in C#*. s. 3.

3.5 Fraktál

Ve své práci pracuji s Mandelbrotovou množinou. Mandelbrotova množina je fraktál neboli geometrický objekt, který je soběpodobný. Fraktál lze nejjednodušeji definovat jako nekonečně členitý útvar. Opakem nekonečně členitého útvaru je geometricky hladký útvar. Příkladem geometricky hladkých útvarů jsou euklidovská tělesa, jako je přímka, kruh, rychle, koule atd. [18]



Obrázek 2 Mandelbrot[19]

3.5.1 Výpočet fraktální množiny

Výpočet fraktálu je dán vzorcem $Z_{n+1} = Z_n^2 + c$ neboli, následující bod je dán mocninou předchozího. Jedná se tedy o posloupnost bodů, pro které platí $|Z_n| > 2$ do množiny nepatří, od posloupnosti diverguje. Pro výpočet fraktálu využívám únikového algoritmu, kde každý bod je počítán v cyklu. Cyklus má danou horní hranici v počtu iterací, které když dosáhne, tak počítaný bod je brán jako divergentní. Tuto horní hranici také můžeme nazývat přesnost algoritmu. Pro nízká čísla (<50) začíná být obrazec rozmazaný. Každému bodu je přidělena barva, která je spočtena z počtu iterací. [20]

---

[18] *Definice Fraktálů* [online]. Dostupné z: http://www.ksr.tul.cz/fraktaly/definice.html.
[19] Obrázek2 vlastní zdroj
[20] *Fraktály* [online]. Dostupné z: http://kmlinux.fjfi.cvut.cz/~pauspetr/html/skola/fraktaly/res.htm#_Toc73066121.

3.6 Faktoriál

Faktoriál je matematická operace, která definovaná pro přirozená čísla, včetně nuly. Značíme ji vykřičníkem za číslem a platí, že: [21]

$$n! = \prod_{k=1}^{n} k$$

Speciálně potom 0! = 1.

3.7 Matice a jejich násobení

3.7.1 Definice matice

„Nechť je dána množina M a čísla m, n ∈ N. Maticí A typu $m \times n$ nad množinou M budeme rozumět obdélníkové schéma

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{pmatrix}$$

kde $a_{ij} \in M$“ [22]

3.7.2 Násobení matic

Pro násobení matic musí platit, že počet sloupců první matice musí být stejný jako počet řádků druhé matice.

„Nechť **A** = $(a_{ij})_{m\,x\,n}$, **B** = $(b_{jk})_{n\,x\,p}$ jsou matice nad okruhem R.

Součinem matic **A,B** rozumíme matici C = $(c_{ik})_{m\,x\,p}$, pro jejíž prvky platí“

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \,.\, b_{jk} \;\forall i = 1,\ldots,m;\; k = 1,\ldots,p.$$

[23]

[21] *Matematika* [online]. Dostupné z: https://matematika.cz/faktorial.
[22] *Matice* [online]. Dostupné z: http://reseneulohy.cz/1296/definice-a-vlastnosti-matice.
[23] *Matice* [online]. Dostupné z: http://reseneulohy.cz/1297/scitani-a-nasobeni-matic.

4 Vlastní práce

4.1 Součet prvků v poli

První aplikace, které se budu věnovat je součet prvků v poli. Jedná se o triviální úlohu, na které se dá ale mnoho demonstrovat. Na této jednoduché úloze lze totiž demonstrovat téměř všechny možnosti paralelního programování. I v úloze jako je tato lze narazit na nějaká úskalí.

4.1.1 Úskalí aplikace

4.1.1.1 Pole vs. generikum

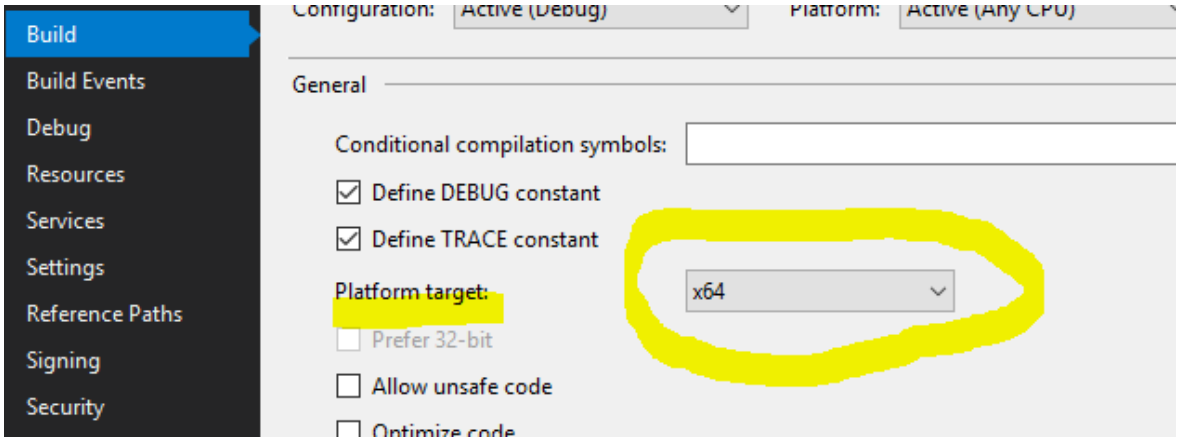
V tomto odstavci trošku předběhnu a budu se věnovat problémům, které mohou nastat při programování této aplikace. Protože jakákoli aplikace, která má efektivně využívat paralelizace potřebuje větší datovou základnu, tak první úskalí, na které zde narazíme, je vytvoření velkého pole. Určitě někoho napadne, proč bych to tedy řešil přes pole, když můžu využít generikum? Pokud bych generikum, v tomto případě asi List<int>, použil tak jako používám pole (inicializace pomocí new a v kostruktoru počet prvků), tak se vůbec nic nezmění a kromě trochu jiných způsobů použití vše bude fungovat. Ovšem pokud bych generikum použil s prázdným konstruktorem, tak si „řekne“ klidně i o dvojnásobek paměti! To už je problém.

4.1.1.2 Paměť

Standardně mi Visual studio nastavilo u aplikace, že se jedná o 32bit aplikaci. To povoluje standardně aplikaci, resp. jednomu objektu aplikace, zabrat 2 GB paměti. To je značně limitující, protože primitivní úloha jako je tako, zabere na výpočet pár jednotek sekund. Proto potřebuji pro relevantní měření opravdu hodně dat. Existují řešení, která toto omezení povolují obejít:

- Nastavit aplikaci jako 64 bitovou.

V nastavení aplikace (properties) v záložce build je možné zvolit platfomu. Viz. obrázek.



Obrázek 3 nastavení aplikace[24]

- Povolit veliké objekty v konfiguraci.
  < gcAllowVeryLargeObjects enabled="true"/>

Toto je možné od .NET4.5. Výsledný konfigurační XML pak bude vypadat například následovně:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true"/>
  </runtime>
</configuration>
```

4.1.1.3 Varování autora

Všechny tyto úpravy mě pomohly k tomu, že šlo vytvořit pole o 10^9 prvků. Větší pole už jazyk nezvládne. Pokud by přeci jen někdo potřeboval větší, tak bude muset využít generikum s prázdným konstruktorem. To pak pomocí metody Add()naplnit. Paměťová náročnost je zde ale už opravdu vysoká. Chtěl bych zde varovat před vytvářením příliš velkých polí. Aplikaci doporučuji testovat přiměřeně k operační paměti systému. Příliš velký požadavek na paměť může způsobit pád systému a nebo modrou smrt.

[24] Obrázek 3 vlastní zdroj

4.1.2 Příprava dat

Vše se děje v jedné třídě, kterou jsem nazval SumClass. V kostruktoru třídy se vytvoří pole které obsahuje přesně tolik prvků kolik si uživatel poručí. Toto pole se pak pomocí metody void Generator() naplní náhodnými čísly od do podle konstanty.

```
var rnd = new Random();
for (int i = 0; i < _pocetPrvku; i++)
{
  _pole[i] = rnd.Next(RndDo);
}
```

4.1.3 Klasické řešení

Zde bych rád podotknul že existuje více řešení. První je klasickým cyklem `for` a nebo `foeach`. Já doporučuji požívat `foeach`, protože je zde eliminována možnost chyby v indexu. Druhé řešení, které je mnohem „elegantnější“ je požít metodu .Sum() dotazovacího jazyka LINQ. Ovšem zde se musíme mít na pozoru, protože mezitím, co cyklus nám dovoluje přetečení integeru, LINQ to nedovoluje a vyvolá výjimku. Z toho lze vyvodit další důsledek a to je, že ač generátor pracuje pouze s kladnými čísly, výsledek může díky přetečení vyjít záporný. Samotný algoritmus pak vypadá takto:

```
foreach (int i in _pole)
{
 Soucet += i;
}
```

4.1.4 Řešení pomocí Tasks

Řešení spočívá v tom, že datovou základnu rozdělím na půl a každou půlku budu sčítat v separátním tasku. Velice jednoduché a efektivní řešení pro danou úlohu. K úloze se váže sekundární funkce, která sčítá prvky v poli od zadaných mezí po zadané meze.

```
private int Suma(int od, int kam) // funkce na paralelní řeš. -> rozlišuje meze
{
  int vysledek = 0;
  for (int j = od; j < kam; j++)
  {
    vysledek += _pole[j];
  }
  return vysledek;
}
```

Samozřejmě, že by se pole dalo rozdělit na více než půlky, ale já se rozhodl, že pro danou úlohu a demonstraci nám půlky stačí.

```
Soucet = 0;
int pocet = _pole.Length;
int interval = pocet / 2; //2 vlakna
int zbytek = pocet % 2;
```

```
var ukoly = new Task<int>[2];
ukoly[0] = new Task<int>(() => Suma(0, interval));
ukoly[0].Start();
ukoly[1] = new Task<int>(() => Suma(interval, interval * 2 + zbytek));
ukoly[1].Start();
Soucet = ukoly[0].Result + ukoly[1].Result;
```

4.1.5 Řešení pomocí paralelního cyklu

Toto řešení už není tak jednoduché a přímočaré jako pomocí tasků. Ukáži dvě řešení, kdy v prvním dojde k datarace (křížení dat) a tím pádem k špatným výsledků. Vyřešení datarace použiju lock, což bude mít za následek katastrofické zpomalení a následné zamítnutí daného řešení. Druhé řešení už bude lepší a bude realizováno pomocí anonymní funkce a interního mechanismu paralelního cyklu. Toto řešení také využívá lock ovšem už je méně časově náročné než předchozí.

První řešení

```
Parallel.For(0, _pole.Length, i =>
{
  lock (sync)
  {
    int s = Soucet;
    Soucet = Secti(s, _pole[i]);
  }
});
```

Toto řešení by se dalo slovně interpretovat asi takto, od nuly, do počtu prvků v poli pro každé i vykonej.

Druhé řešení

Vylepšení prvního řešení, ovšem stále neefektivní.

```
Parallel.For(0, _pole.Length, () => 0, (vysl, pls, mezivysl) =>
{
  return mezivysl += _pole[vysl];
}, (lhod) =>
  {
    lock (sync)
    {
      Soucet += lhod;
    }
 });
```

4.1.6 Řešení pomocí Parallel.Invoke

Řešení je naprosto totožné jako pomocí tasků, s tím rozdílem, že jedna půlka dat je vypočtena v hlavní vlákně a druhá na pozadí.

```
int pocet = _pole.Length;
```

```
int interval = pocet / 2;
int zbytek = pocet % 2;
int vysl1;
vysl1 = 0;
Parallel.Invoke(() => vysl1 = Suma(0, interval));
int vysl2 = Suma(interval, pocet + zbytek);
Soucet = vysl1 + vysl2;
```

Opět triviální a velmi efektivní řešení.

4.1.7 Testy

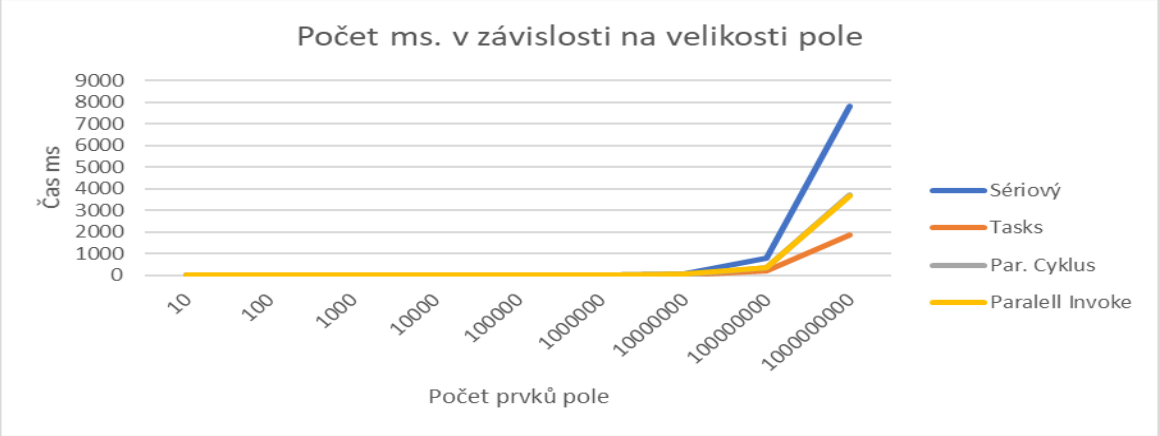
Pro účely testování jsem napsal zvláštní třídu. Tako třída obsahuje 2 metody, první metoda pro opakované testování metody a druhá pro zápis časových hodnot testů do souboru. Kvůli lepší přehlednosti a následné práci s daty jsem zvolil soubor typu CSV. Výsledné časy pak jednoduše pomocí tabulkového editoru z průměruji a stanovím závěry.

	Velikost pole	Sériový[ms]	Tasks[ms]	Par. Cyklus[ms]	Paralell Invoke[ms]
1	10	0,003304	0,19581	0,284745	0,008129
2	100	0,00151	0,02998	0,010955	0,001976
3	1000	0,008854	0,027739	0,039362	0,006354
4	10000	0,077956	0,058443	0,10817	0,038011
5	100000	7,866802	1,906808	3,854175	3,556403
6	1000000	7,731137	1,873391	3,722073	3,568452
7	10000000	80,77757	18,735286	31,758004	36,69732
8	100000000	786,3893	187,250444	366,221117	367,042289
9	1000000000	7836,704	1849,126696	3731,91239	3658,337001

Tabulka 1 testy sčítaní prvků v poli[25]

V tabulce lze vyčíst hodnoty průměrného času vykonání úlohy sčítání na různých velikostech pole. Hodnoty pod jednotlivými metodami jsou v milisekundách. Z hodnot lze jednoznačně vyčíst přínosnost paralelního řešení oproti klasickému. Přínos paralelizace pak lze pozorovat i na následujícím grafu.

[25] Tabulka 1 vlastní zdroj



Obrázek 4 graf časové závisloti úlohy na velikosti pole[26]

4.1.8 Komentář a hodnocení výsledků

Tato úloha jednoznačně prokazuje výhodnost paralelizace. Pro pole do 10tis. prvků je paralelní algoritmus pomalejší kvůli režii na paralelizaci, ovšem právě v těch 10tis. se to láme a paralelní algoritmus začíná být výhodnější. Na grafu lze pozorovat, že rozdělení úlohy mezi dva tasky přináší nejlepší zrychlení. Paralelní cyklus se překrývá s paralell.invoke, oba jsou tedy stejně časově nároční. Avšak paralelní cyklus pro tuto úlohu nedoporučuji, protože na rozdíl od p.invoke, cyklus využívá veškerý, procesorové prostředky (využití cpu = 100%) a tím pádem je mnohem méně efektivní. Toto chování je způsobené zámkem prostředků lock, proto doporučuji se mu vyvarovat. Pokud bych měl porovnat nejrychlejší (task) a nejpomalejší (sériový), tak při testování, kdy 100krát provedu úlohu na poli s 10^9 prvků, tak sériový přístup zabral 13minut a task 3 minuty.

4.2 Paralelní násobení matic

Protože v předchozím příkladu paralelní cyklus svojí efektivitou selhal, rozhodl jsem se pro demonstraci algoritmu, kterému je paralelní cyklus jako na míru šitý. U tohoto programu narazíme jako u předchozího na paměťové limity pole. Ovšem zde se spokojím s omezením do 10000 prvků, protože algoritmus už je náročnější než u předešlého příkladu. Program tedy obsahuje generátor velmi podobný předchozímu příkladu. Listbox který používám pro zobrazení výsledků je pouze informativní pro malé matice. Větší už nezvládne zobrazit a doporučuji ho v kódu za komentovat.

[26] Obrázek 4 vlastní zdroj

4.2.1 Ovládání programu

Tento příklad již využívá uživatelské rozhraní Windows Forms. Je to z důvodu, že jsem chtěl zde chtěl demonstrovat komponentu BackgroundWorker. Tato komponenta je velice jednoduchý nástroj, pomocí jehož můžeme zamezit nepříjemnému zamrznutí aplikace při vykonávání časově náročné operace. Svým způsobem je to tedy také forma paralelizace, protože oddělíme část kódu od hlavního vlákna, které obsluhuje uživatelské rozhraní.

4.2.2 BackgroundWorker

Tuto komponentu do našeho projektu dostaneme jednoduše přetažením z toolboxu Visual studia. Komponenta nám nabízí tři události:

1. DoWork událost

Událost, resp. v těle této události se vykonává kód. Kód je vykonáván na jiném vlákně, něž běží naše UI. Z tohoto důvodu je důležité si dávat pozor na to, jaké komponenty UI budeme z této metody aktualizovat. Komponenty, které nejsou thread save musí být ošetřeny.

2. ProgressChanged událost

Tato událost je užitečná, pakliže potřebujeme uživatele informovat o změně, pokročení v procesu vykonávání.

3. RunWorkerComplete

Blok kódu v této události se provede po skončení běhu workeru. Jak jsem uvedl, některé komponenty nelze aktualizovat z jiného vlákna, proto můžeme využít tuto událost a aktualizovat je po vykonání kódu ve DoWork.

4.2.3 Klasické řešení

Dle matematického postupu popsaného v teoretické části práce, provedu skalární součiny.

```
for (int row = 0; row < MatA.GetLength(0); row++)
{
    for (int col = 0; col < MatB.GetLength(1); col++)
    {
      for (int inner = 0; inner < MatA.GetLength(0); inner++)
      {
        Result[row, col] += MatA[row, inner] * MatB[inner, col];
      }
    }
}
```

4.2.4 Paralelní řešení – pomocí cyklu

```
Parallel.For(0, MatA.GetLength(0), row =>
{
       for (int col = 0; col < MatB.GetLength(1); col++)
    {
        for (int inner = 0; inner < MatA.GetLength(1); inner++)
        {
            Result[row, col] += MatA[row, inner] * MatB[inner, col];
        }
    }
});
```

Protože výpočty nejsou závislé na předchozích krocích, paralelní cyklus zde využije svůj plný potenciál.

4.2.5 Použití BackgroudWorker

BackgroudWorker spustíme `backgroundWorker1.RunWorkerAsync();`. Abychom nemuseli mít BackgroudWorker komponent stejný počet jako funkcí které chceme oddělit od hlavního vlákna, doporučuji předat funkci Workeru odkazem. Tělo Workeru pak může vypadat následovně:

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
   _timer.Reset();
   _timer.Start();
     _testClass.TestFuntion(_act);
   _act();
   _timer.Stop();
}
```

4.2.6 Zrušení vykonávání

Protože je tato úloha časově náročná, chtěl bych zde demonstrovat způsob, jak zrušit vykonávání úlohy běžící na pozadí. Nástrojem pro zrušení vykonávání je: `CancellationTokenSource _tokenSource;`

Od něho si vytvoříme instanci, a token této instance předáme Tasku. V průběhu vykonávání je pak potřeba neustále kontrolovat, jestli není požadavek na zrušení.

```
if(ct.IsCancellationRequested)
  ct.ThrowIfCancellationRequested();
```

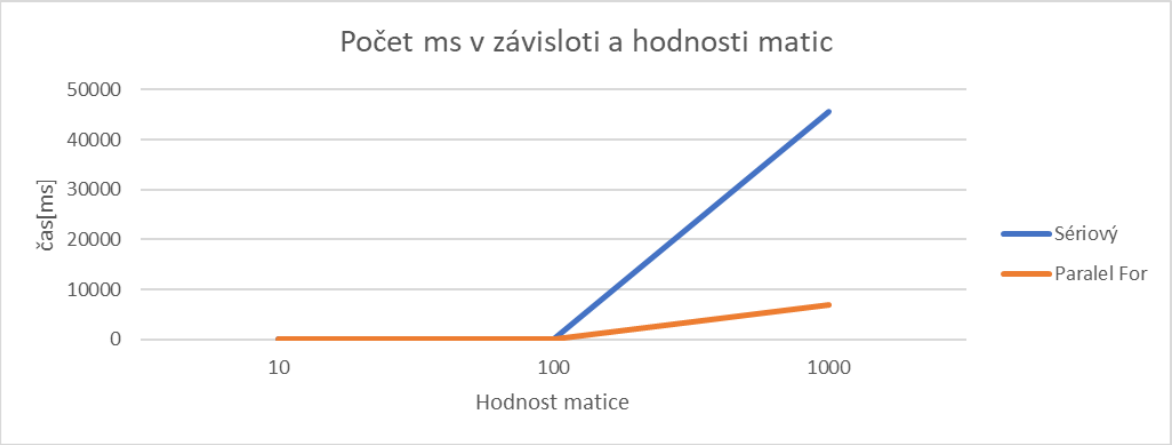
V tomto příkladu jsem využil cyklů, které provádějí násobení. Pak už stačí jen zavolat `_tokenSource.Cancel();` a vykonávaný kód se přeruší.

4.2.7 Testy

Pro testování jsem použil stejnou třídu jako v předchozím příkladu. Výsledky testů dopadly následovně:

	Hodnost matice	Sériový	Paralel For
1	10	0,04391	0,181786
2	100	37,4805	7,227894
3	1000	45697,5	6978,3004

Tabulka 2 testy násobení matic[27]



Obrázek 5 graf časové závislosti na hodnosti matice[28]

4.2.8 Komentář a hodnocení výsledků

Z tabulky můžeme vyčíst, že pro malé matice je sériový přístup o pár milisekund rychlejší. Čím je ale hodnost matice vyšší, tím více dramaticky narůstá zisk paralelizací. Paralelní cyklus je tady opravdu velmi účinný. Při měření matice 1000x1000, 100 měření sériového přístupu trvalo zhruba jednu a čtvrt hodinu. Paralelnímu cyklu to při těch samých podmínkách trvalo necelou čtvrt hodinu.

4.3 Paralelní faktoriál

Faktoriál je oblíbená úloha na rekurzi. Protože výpočtem faktoriálu mohou vznikat opravdu velká čísla, výpočet velkého faktoriálu by mohla být zajímavá úloha na paralelizaci.

[27] Tabulka 2 vlastní zdroj
[28] Obrázek 5 vlastní zdroj

4.3.1 Úskalí úlohy

Každý, kdo někdy zkoušel naprogramovat faktoriál zjistil, že výpočet velmi rychle vyčerpá limit velikosti proměnné. U proměnné int je maximální faktoriál 12 u proměnné long je to potom 20. To jsou pro nás nezajímavé hodnoty. Naštěstí Microsoft i na tyhle případy myslel a připravil knihovnu Numerics. Je zapotřebí ji ručně do projektu naimportovat a poté můžeme využívat BigInteger. Ten umí opravdu dostatečně velká čísla. Ovšem i s touto knihovnou se vyhnu rekurzi, protože pro větší hodnoty faktoriálu přeteče zásobník a program skončí s výjimkou overflow.

4.3.2 Klasické řešení

Jak jsem napsal, z důvodu výše uvedeného, se vyhnu rekurzi. Kód tedy vypadá následovně:

```
BigInteger output = 1;
while (n > 0)
{
    output = output * n;
    n--;
}
return output;
```

4.3.3 Paralelní řešení

Ne vždy je úplně efektivní data nějakým způsobem dělit mezi dostupné prostředky procesoru. Operace dělení zabere nějaký čas a taky zde vzniká prostor pro případné chyby. Doporučuji tedy se před realizací zamyslet a zkusit přijít na jiné řešení, které bude elegantnější a efektivnější. Faktoriál jsou mezi sebou násobená čísla, kde nezáleží na tom, v jakém pořadí je mezi sebou vynásobíme. První myšlenka, která mě napadla byla, vynásobit mezi sebou lichá a sudá čísla a výsledky z obou součinů poté vynásobit mezi sebou. Řešení bylo funkční a efektivní, ale dá se ještě zdokonalit. Co takhle mezi sebou násobit n + počet tasků prvků? Výsledný kód by pak vypadal takto:

```
var taskCount = Environment.ProcessorCount;
var tasks = new Task<BigInteger>[taskCount];
for (var i = 0; i < taskCount; i++)
    tasks[i] = Task<BigInteger>.Factory.StartNew(vstup =>
    {
        var citac = (BigInteger) vstup;
        var multiplikace = new BigInteger(1);
        while (citac <= n)
        {
            multiplikace = multiplikace * citac;
            citac = citac + taskCount;
        }
        return multiplikace;
```

```
    }, new BigInteger(i + 1));
var vystup = new BigInteger(1);
Task.WaitAll(tasks);
foreach (var uloha in tasks)
    vystup = vystup * uloha.Result;
return vystup;
```

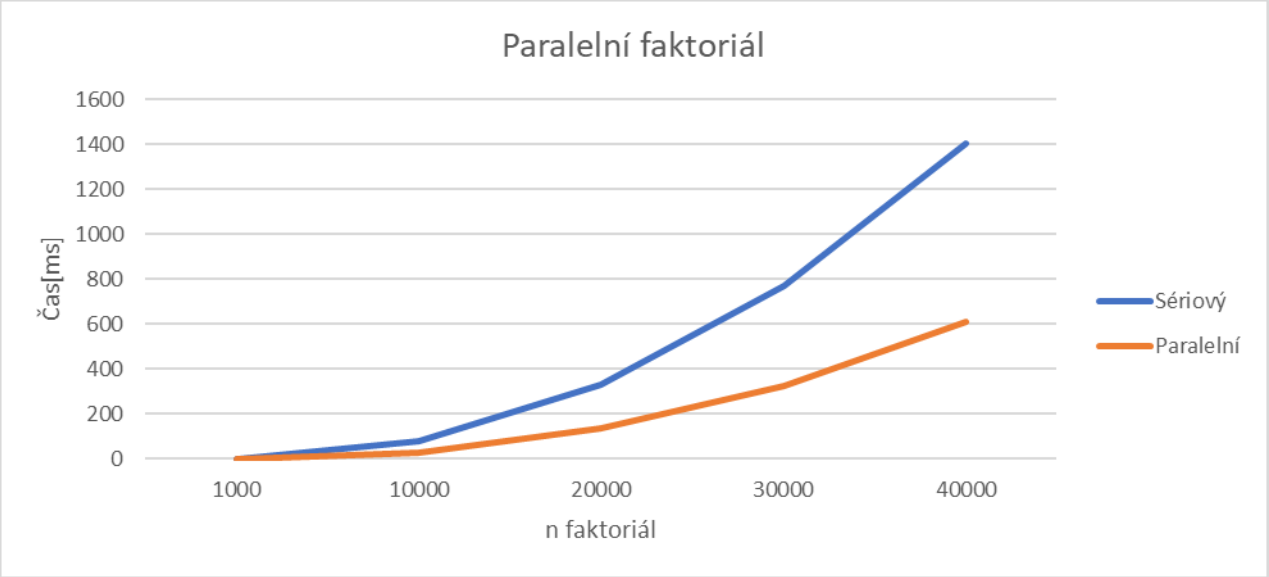
4.3.4 Testy

Pro testování opět využiji testovací třídu z příkladu 1. jen musím upravit hlavičku testovací funkce. Pro účely testování opět doporučuji za komentovat výstup do richtexboxu.

```
public void TestFuntion(Func<BigInteger, BigInteger> action, BigInteger n)
```

	Sériový	Paralelní
1000	0,759594	0,459584
10000	78,26045	29,11001
20000	329,1653	134,8
30000	771,0302	327,8959
40000	1401,896	608,7213

Tabulka 3 faktoriál[29]



Obrázek 6 graf závislosti času na n faktoriál[30]

4.3.5 Komentář a hodnocení výsledků

Tato úloha začíná v porovnání s klasickým vykonáváním zrychlovat až od faktoriálu jednotek tisíc. Zvolil jsem zde jemnější odstupňování a tím pádem se na grafu krásně promítlo zrychlení paralelizaci algoritmu.

4.4 Mandelbrotova množina

Fraktální množiny jsou oblíbené jako algoritmy co se týče demonstrace paralelizace. K jejich paralelizaci není zapotřebí ani tak vysoká matematika jako spíš abstrakce myšlení nad daným problémem. Algoritmus jako takový je poměrně rychlý (velmi rychle konverguje) a tak hlavní optimalizace nastávají při vykreslování.

4.4.1 Popis programu

Fraktální množinu jsem naprogramoval tak, že na počátku (klik na tlačítko nový) se vykreslí fraktální množina v počátečních bodech. Kliknutím na obrázek se pak množina

[29] Tabulka 3 vlastní zdroj
[30] Obrázek 6 vlastní zdroj

přiblíží v místě kliknutí. Při přibližování je krásně vidět, jak se množina sebe opakuje. V programu je pak pomocí RGB palety rozlišováno, které body do množiny patří a které ne. Počet iterací vlastně určuje přesnost, s kterou se množina vypočte a vykreslí.

4.4.2 Faktory ovlivňující rychlost

4.4.2.1 Počet iterací

Algoritmus konverguje v celku rychle a ta i při pouhých 50 iteracích už můžeme pozorovat základní tvary množiny. Ovšem při přibližování na 50 iteracích už můžeme pozorovat nepřesnosti. Kolik iterací teda zvolit? Počet iterací bych volil podle zařízení, na kterém bych chtěl vykreslovat. Na mobilech nebo jiných zařízeních bych volil počet iterací kolem 500. Už při 100 je přesnost celkem vysoká, ale z vyšších iterací můžeme pozorovat, jak při nižších máme chudší barevnou paletu obrazu. Na počítačích bych viděl ideální počet iterací kolem 1000. Tisíc iterací jsem tedy nastavil jako základní hodnotu u svého programu. Desítky tisíc iterací postrádá smysl, vykreslování se zpomalí, ale výsledná přesnost je dána přesností proměnné double.

4.4.2.2 Rozlišení

Druhým velice důležitým faktorem pro rychlost vykreslení je rozlišení. Čím větší je rozlišení vykreslované množiny, tím více času budeme potřebovat.

4.4.3 Klasické řešení

Pro naprogramování množiny jsem se inspiroval v pseudo-kódu na anglické wikipedii. https://en.wikipedia.org/wiki/Mandelbrot_set Převod do jazyka c# pak vypadá následovně:

```
public virtual Bitmap Mandel()
{
    double dx, dy, cx, cy;
    Bitmap img = new Bitmap(Width, Height);
    double posunx = (Maxr - Minr) / Width;
    double posuny = (Maxi - Mini) / Height;
    double tempzx;

    for (int x = 0; x < Width; x++)
    {
        cx = posunx * x - Math.Abs(Minr);
        for (int y = 0; y < Height; y++)
        {
            dx = 0;
            dy = 0;
            cy = posuny * y - Math.Abs(Mini);
            int count = 0; // pocet iteraci
```

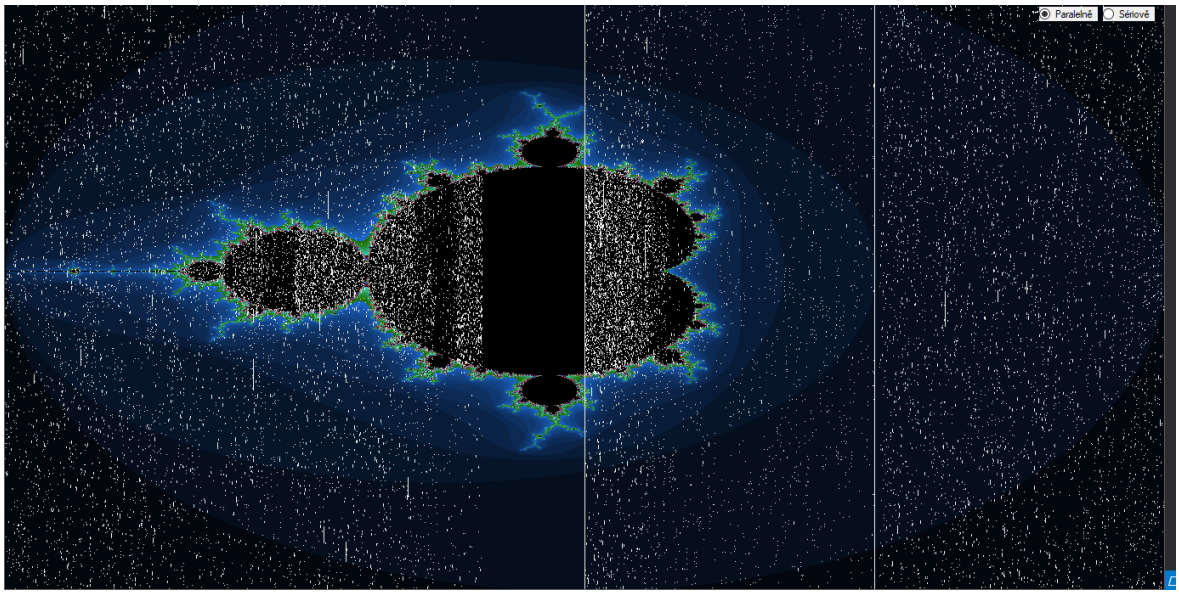
```
            while (dx * dx + dy * dy <= 4 && count < IterationCount) /*Cyklus který
určí krajní body a body které patří do množiny*/
            {
                count++;
                tempzx = dx;
                dx = dx * dx - dy * dy + cx;
                dy = 2 * tempzx * dy + cy;
            }
            img.SetPixel(x, y, count != IterationCount ? Color.FromArgb(count % 128
* 2, count % 32 * 7, count % 16 * 14) : Color.Black);
        }
    }
    return img;
}
```

4.4.4 Paralelní řešení

Při paralelizaci sekvenčního řešení mě napadlo více řešení, z nichž některé nevedly k cíli nebo k cíli vedly, ale nedosáhl jsem takového zrychlení, jaké bych chtěl. Pro demonstraci předvedu i tato řešení.

4.4.4.1 Paralelizace s přímým vykreslováním do bitmapy

4.4.4.2 Řešení spočívá v tom, že si počítání rozdělím na příslušné intervaly, které přerozdělím mezi jednotlivé Tasky. Uvnitř těla tasku pak budu zapisovat jednotlivé pixely do bitmapy. Toto řešení vyvolá výjimku. Když místo výjimky obalíme blokem try výsledný obrazec bude vypadat asi takto.



Obrázek 7 Mandelbrot s datarace[31]

[31] Obrázek 7 vlastní zdroj

Co se vlastně stalo špatně? Více vláken najednou se pokusilo najednou zapsat do stejného obrázku, který ovšem v té době nepodporuje zápis více vláken najednou. V době zápisu prvního vlákna se obrázek zablokoval pro zápis a když se další vlákna pokoušela zapsat tak nemohla a vznikl prázdný prostor. Toto se dá ošetřit zámkem procesu a výsledný kód by pak vypadl takto:

```
Bitmap img = new Bitmap(Width, Height);
object zamek = new object();
Task[] ukoly = new Task[_tasks];

for (int i = 0; i < _tasks; i++)
{
    int i1 = i;
    ukoly[i] = new Task(() =>
    {
        double dx;
        double dy;
        double cx;
        double cy;
        double posunX = ((Maxr - Minr) / Width);
        double posunY = ((Maxi - Mini) / Height);
        for (int x = (i1 * Width / _tasks); x < (Width / _tasks * (i1 + 1)); x++)
        {
            cx = (posunX * x) - Math.Abs(Minr);
            for (int y = 0; y < Height; y++)
            {
                dx = 0;
                dy = 0;
                cy = (posunY * y) - Math.Abs(Mini);

                // špatná myšlenka - zámek algoritmus šíleně zpomalí
                lock (zamek)
                {
                    int loopgo = Pixel(dx, dy, cx, cy);
                    Color c = Color.FromArgb(loopgo % 128 * 2,
                        loopgo % 32 * 7, loopgo % 16 * 14);
                    img.SetPixel(x, y, c);
                }
            }
        }
    });
    ukoly[i].Start();
```

Toto řešení má však zásadní problém a to, že je velmi pomalé, pomalejší než sériový přístup. Tato paralelizace se tedy minula účinkem.

4.4.4.3 Paralelizace s výsledným vykreslením

Předchozí myšlenku je potřeba vylepšit. To mě tedy vedlo k nápadu oddělit vykreslení od výpočtu. Výpočet se provede ve vláknech do pole výsledků a to se následovně vykreslí. Taková to lehká úprava způsobí, že algoritmus už zrychlí nad úroveň sériového.

```
Bitmap img = new Bitmap(Width, Height);
```

```
object zamek = new object();
Task[] ukoly = new Task[_tasks];
int[,] pole = new int[Width, Height];
for (int i = 0; i < _tasks; i++)
{
int i1 = i;
    ukoly[i] = new Task(() =>
    {
        double dx;
        double dy;
        double cx;
        double cy;
        double posunX = ((Maxr - Minr) / Width);
        double posunY = ((Maxi - Mini) / Height);
        for (int x = (i1 * Width / _tasks); x < (Width / _tasks * (i1 + 1)); x++)
        {
            cx = (posunX * x) - Math.Abs(Minr);
            for (int y = 0; y < Height; y++)
            {
                dx = 0;
                dy = 0;
                cy = (posunY * y) - Math.Abs(Mini);
                pole[x, y] = Pixel(dx, dy, cx, cy);
            }
        }
    });
    ukoly[i].Start();
}
Task.WaitAll(ukoly);
for (int i = 0; i < Width; i++)
{
    for (int j = 0; j < Height; j++)
    {
        img.SetPixel(i, j, Color.FromArgb(pole[i, j] % 128 * 2, pole[i, j] % 32 * 7,
pole[i, j] % 16 * 14));
    }
}
```

Ovšem i zde je prostor pro zlepšení. Paralelizace s rozdělením bitmapy na proužky. Toto řešení už je implementačně trochu složitější než předchozí, ale výkonově nás posune opět trochu dopředu. Logika je vcelku jednoduchá. Rozdělit algoritmus i kreslení do více Tasků. Na počátku si musíme podle počtu požadovaných Tasků rozdělit obrázek a výpočetní část. Meze jsem stanovil touto funkcí:

```
private int[] Meze(int width, int threads)
{/*funkce vypočte meze pro každý kus obrázku*/
    /*pokud se to dělí přímo v cyklu tak vznikají bílé pruhy (díry) v bitmapě*/
    int w = width / threads;
    int[] meze = new int[threads + 1]; //chci přidat ještě nulu
    meze[0] = 0;
    for (int x = 1; x <= threads; x++)
    {
        meze[x] = x * w;
        if (meze[x] + w >= width)
        {
            meze[x] = meze[x] + (width - meze[x]);
```

```
        }
    }
    return meze;
}
```

Výpočetní algoritmus je poté stejný, s tím rozdílem, že každý Task si kreslí do své bitmapy. Na konci je potřeba z vypočtených bitmap složit výslednou. To se udělá následovně:

```
Graphics g = Graphics.FromImage(_finalBitmap);

foreach (Bitmap image in _images)
{
    g.DrawImage(image, new Rectangle(0, 0, image.Width, image.Height));
}
```

4.4.5 Testy

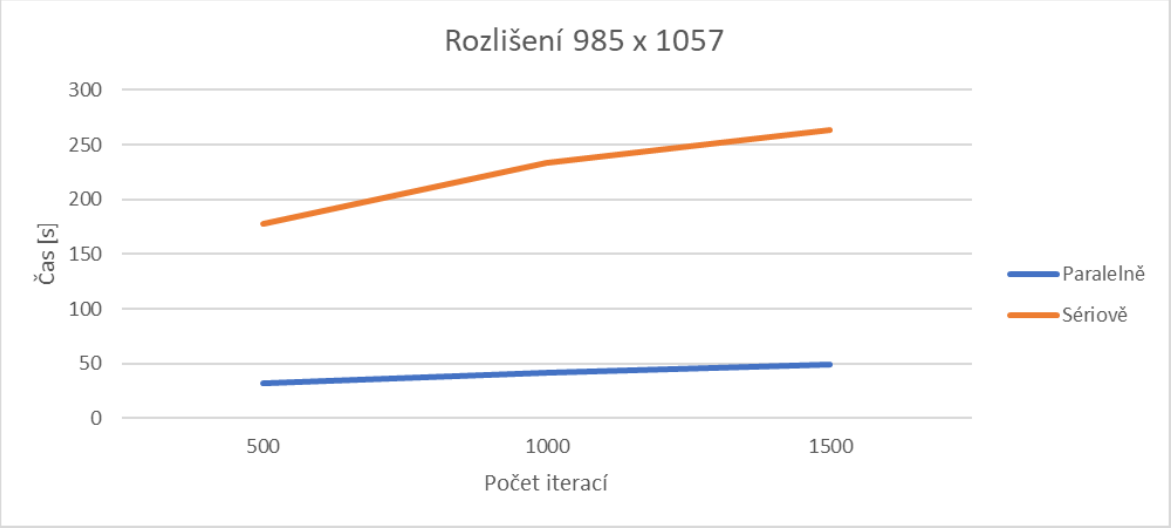
Abych nasimuloval stejné podmínky, nejdříve vykreslím novou množinu a poté do ní budu zoomovat. Body zoomu získám vyklikáním na obrázek. Poté spustím test zoomu a provede se zoom předchozích hodnot. Toto testování provedu pro 3 různá rozlišení s 3 různými počty iterací. Chtěl bych zde podotknout, že bitmapa není ideální nástroj pro rychlé překreslování, a tak se může stát, že při nízkém počtu iterací při rychlém zoomu se nestihne překreslit a způsobí pád aplikace.

4.4.5.1 Nízké rozlišení

Rozlišení	Počet Iterací	Paralelně	Sériově
985 x 1057	500	31,6547	177,6323
	1000	41,7493	233,7363
	1500	49,5563	263,0207

Tabulka 4 nízké rozlišení[32]

[32] Tabulka 4 vlastní zdroj

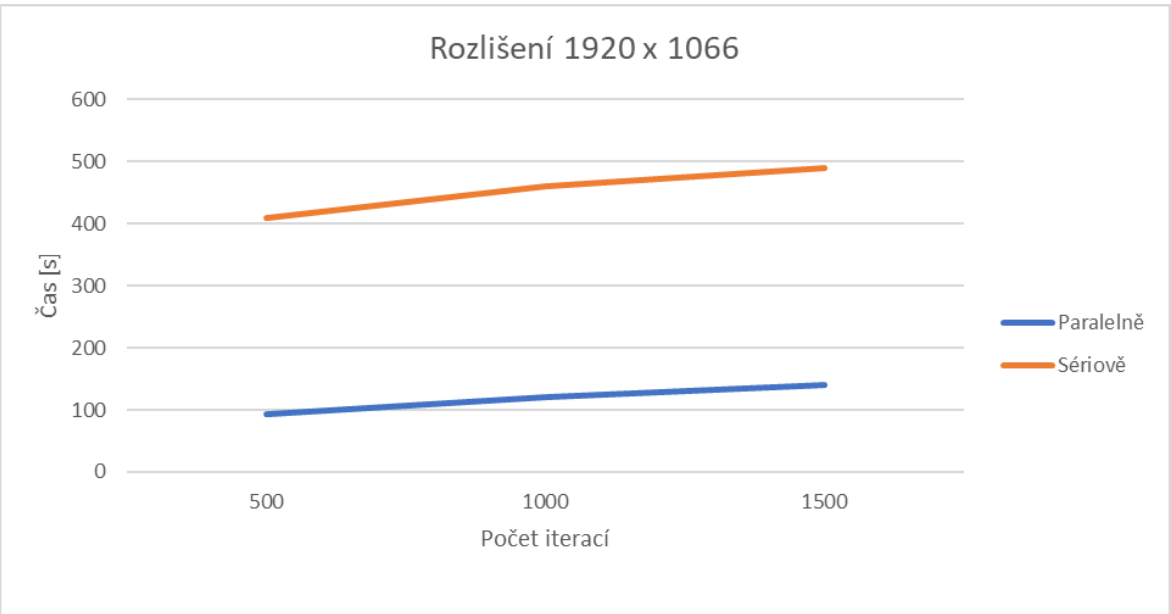


Obrázek 8 graf pro nízké rozlišení[33]

4.4.5.2 Střední rozlišení

Rozlišení	Počet iterací	Paralelně	Sériově
1920 x 1066	500	92,6137	408,2517
	1000	120,5569	460,3563
	1500	140,6515	490,3356

Tabulka 5 střední rozlišení[34]



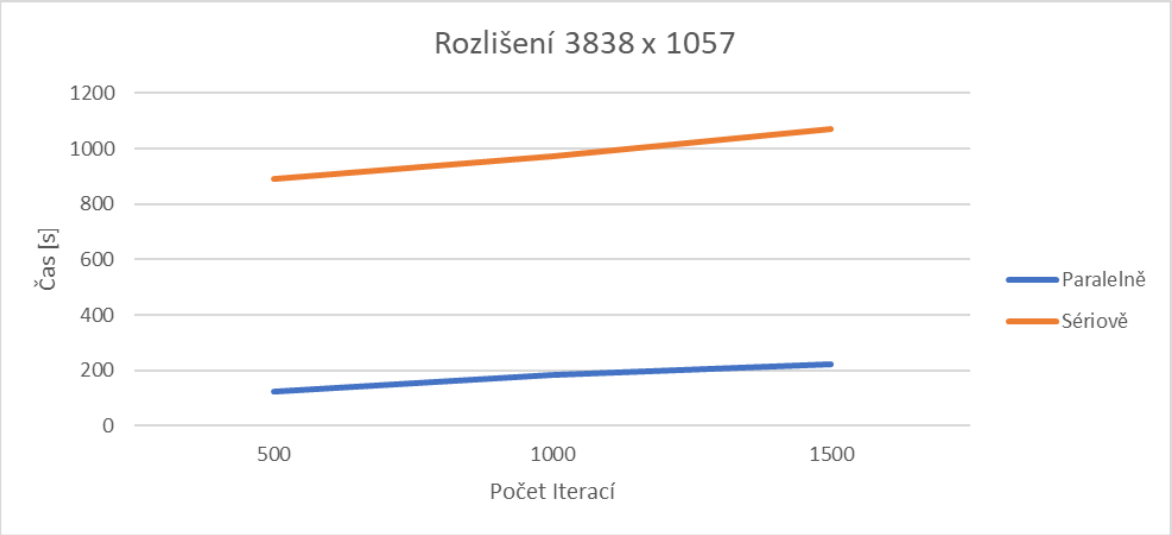
Obrázek 9 graf pro střední rozlišení[35]

[33] Obrázek 8 vlastní zdroj
[34] Tabulka 5 vlastní zdroj
[35] Obrázek 9 vlastní zdroj

4.4.5.3 Vysoké rozlišení

Rozlišení	Počet iterací	Paralelně	Sériově
3838 x 1057	500	122,5233	890,5
	1000	182,4879	971,0223
	1500	222,5033	1070,067

Tabulka 6 vysoké rozlišení[36]



Obrázek 10 graf pro vysoké rozlišení[37]

[36] Tabulka 6 vlastní zdroj
[37] Obrázek 10 vlastní zdroj

5 Závěr

Za cíl práce jsem si stanovil ukázat na aplikacích možnosti paralelizace pomocí Task Parallel Library a ukázat pro které problémové oblasti využití paralelizace přináší výhody a pro které nikoliv. Ve své práci jsem popsal 4 aplikace, v každé z aplikací jsem se snažil ukázat a popsat, jak jsem postupoval a jak jsem přemýšlel, když jsem danou problematiku řešil.

Jako první příklad jsem záměrně zvolil sčítání prvků v poli. Je to triviální úloha, která umožňuje mnoho řešení a tak je na ní možno ukázat skoro všechny paralelní konstrukce. V této úloze selhal paralelní cyklus, protože využíval takřka 100 % cpu, ale výkonově to nebylo znát. Paralelní cyklus ovšem velmi dobře zafungoval v úloze násobení matic. Kvůli tomu jsem tuto úlohu do své práce zahrnul. Z grafu je pak velmi patrné, jak velký rozdíl zde je od klasického sériového kódu. V této úloze jsem také ukázal komponentu backgroundworker, která umožnuje zpracovávat kód na pozadí, aniž by programátor něco věděl o paralelizaci.
Další úloha je Faktoriál. Zde jsem se nejdříve rozhodoval, jestli programovat prvočísla anebo faktoriál. Nakonec faktoriál zvítězil z důvodů zajímavosti velkých čísel a optimalizace výpočtu. Zprvu jsem si chtěl programovat vlastní třídu na velká čísla, ale nakonec jsem v MSDN objevil BigInteger, který můj problém vyřešil. Pak už se jen stačilo zamyslet nad tím, jakým způsobem rozdělit výpočet mezi vlákna, aby to bylo co nejefektivnější.

Poslední příklad je Mandelbrotova množina. Nad touto úlohou jsem strávil nejvíce času. I tak úloha není dokonalá a její největší kámen úrazu je bitmapa. Kvůli špatné správě paměti bitmapy může program zahlásit výjimku. V kapitole o této úloze jsem popsal metodiku, jak jsem postupoval a čemu je lepší se vyhnout. Výsledný paralelní kód úlohu citelně urychlil.

6 Seznam použitých zdrojů

Tištěné zdroje

FREEMAN, Adam. *Pro .NET 4 Parallel Programming in C#*. New York, NY: Apress, 2010. ISBN 978-1-4302-2967-4.

NAKOV & CO., 2013, Svetlin. *FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#*. Sofia, 2013. ISBN 978-954-400-773-7.

Internetové zdroje

Fraktální množiny. *Fraktály* [online]. [cit. 2018-02-11]. Dostupné z: http://kmlinux.fjfi.cvut.cz/~pauspetr/html/skola/fraktaly/res.htm#_Toc73066121

Fraktály. *Fraktální geometrie* [online]. [cit. 2018-03-07]. Dostupné z: http://www.ksr.tul.cz/fraktaly/definice.html

Itnetwork. *Itnetwork* [online]. [cit. 2018-02-04]. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

Definice matice. *Matice* [online]. [cit. 2018-02-17]. Dostupné z: http://reseneulohy.cz/1296/definice-a-vlastnosti-matice

Násobení Matic. *Matice.* [online]. Dostupné z: http://reseneulohy.cz/1297/scitani-a-nasobeni-matic

Faktoriál. *Matematika* [online]. [cit. 2018-02-15]. Dostupné z: https://matematika.cz/faktorial

MSDN. *MSDN* [online]. [cit. 2018-03-07]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/dd460717(v=vs.110).aspx

MSDN. *MSDN* [online]. [cit. 2018-02-04]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

MSDN. *MSDN*[online]. [cit. 2018-02-04]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms173171(v=vs.110).aspx

Parallel Computing [online]. [cit. 2017-12-10]. Dostupné z: https://msdn.microsoft.com/cs-cz/vstudio/bb964701

Intel. *Hyper threading.* Dostupné z: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

7 Přílohy

CD se zdrojovými kódy.