Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Programování s vlákny v jazyce C#

Vít Hlaváček

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Vít Hlaváček

Informatika

Název práce

Programování s vlákny v jazyce C#

Název anglicky

Multi-thread programming in C#

Cíle práce

Bakalářská práce je zaměřena na problematiku programovaní s vlákny v jazyce C#. Hlavními cíli je popsat možnosti vývoje aplikací běžících ve více vláknech a primárně je demonstrovat prostřednictvím ukázkové aplikace.

Metodika

Metodika řešené práce je založena na analyticko-syntetickém přístupu. Bude provedena analýza odborných informačních zdrojů souvisejících s tématem práce a na základě syntézy takto zjištěných poznatků budou shrnuty možnosti práce s vlákny v jazyce C#. Dále bude navržena a implementována ukázková aplikace, která bude pro svůj běh využívat více vláken. Postup implementace bude popsán a zhodnocen.

Doporučený rozsah práce

35-40 stran

Klíčová slova

vlákna, programování, thread, threading, sync, synchronizace, C#

Doporučené zdroje informací

Microsoft Developer Network. C# Programmer's Reference [online]. Dostupné z: https://msdn.microsoft.com/en-us/library/618ayhy6(v=vs.71).aspx

RINGLER, Rodney, 2014. C# multithreaded and parallel programming. ISBN 978-1-84968-832-1.

SHARP, John a John JAGGER, 2002. Microsoft Visual C♯ .NET krok za krokem. Praha: Mobil Media. ISBN 978-80-86593-27-2.

STELLMAN, Andrew a Jennifer GREENE, 2013. Head first C#. Third edition. Beijing: O'Reilly. Head first. ISBN 978-1-4493-4350-7.

Předběžný termín obhajoby

2016/17 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 15. 03. 2017

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Programování s vlákny v jazyce C#" 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 15.3.2017 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi za vedení, pomoc a trpělivost při zpracování této bakalářské práce a také své rodině za podporu a teplé jídlo.

Programování s vlákny v jazyce C#

Souhrn

Práce je zaměřena na navržení vícevláknové aplikace v jazyce Microsoft C#, která generuje fraktál zvaný Mandelbrotova množina a využívá k tomu nové funkce paralelizace .NET Frameworku. Kalkulace mohou probíhat jak paralelním tak synchronním algoritmem.

Program dokáže vizualizovat aproximovanou podobu tohoto fraktálu a měřit rychlost jeho generování v závislosti na vstupních parametrech, jako je úroveň detailů.

Klíčová slova: programování, C#, C sharp, .NET, vlákna, thread, threading, paralelní, vývoj, software, task, TPL, fraktál, mandelbrot

Multi-thread programming in C##

Summary

Aim of this thesis is to develop multithreaded application in Microcoft C# .NET programming language, which will generate fractal called Mandelbrot set and it will make use of a new parallelism methods of .NET framework. Calculations are done with both parallel and serial algorithm.

The program is able to visualize an approximation of that fractal and measure the speed of its calculation, depending on input parameters like depth of detail.

Keywords: programming, C#, C sharp, .NET, thread, threading, parallel, development, software, task, TPL, fractal, mandelbrot

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Vývoj software je dnes potřeba snad pro většinu odvětví lidské činnosti, nejen proto se mohou požadavky, parametry a s nimi spjatá náročnost programů diametrálně lišit. Jedním z mnoha důležitých rozhodnutí, která jsou potřeba při nebo před vývojem učinit, je vybrat vhodný výpočetní model – zda provádění programu rozdělit a popřípadě na kolik částí, vláken, tasků.

Proč je paralelismus důležitý? Růst výpočetního výkonu hardwaru měřený v Hz zatím roste či donedávna rostl dle Moorovu zákonu, avšak tento trend nárůstu hrubého výkonu zmenšováním tranzistorů, a tedy jejich zhušťováním se přiblížil pomyslnému stropu, který je daný nejen dostupnými technologiemi, ale i fyzikálními zákony. (Sharp, 2010)

Od roku 2014 je na trhu procesorů dostupná 14nm technologie a 10nm technologie by měla na trh přijít v průběhu roku 2017 (Pirzada, 2015). Avšak kolem 7nm začínají problémy s kvantovou fyzikou, kdy můžou malé částice jako elektrony projít přes extrémně tenké zdi, aniž by je rozbily. Tento fenomén se nazývá kvantové tunelování. Pominu-li možnosti budoucích technologií, umožňujících nadále zmenšovat velikost tranzistorů, logickým krokem zvýšení výkonu, je spuštění nesekvenčních částí programu najednou – paralelně.

V této práci bude navržen program na generování fraktálu, což vyžaduje algoritmus s takovými vzájemně nezávislými matematickými operacemi, které jsou vhodné k demonstraci vhodnosti použití nejmodernějšího přístupu k paralelnímu programování v prostředí Microsoft C# 6.0 ~ .NET Framework 4.6 na rozdíl od sériovéhu přístupu.

2 Cíl práce a metodika

2.1 Cíl práce

Úkolem této práce je návrh vícevláknové aplikace pomocí programovacího jazyka Microsoft C#. Důraz je přitom kladen na implementaci paralelizace kódu. Algoritmus bude generovat body Mandelbrotovy množiny, a tak bude dostatečně komplexní na to, aby na něm bylo možné demonstrovat význam paralelního programování oproti sériovému.

Část práce bude také věnována implementaci grafickému zobrazení tohoto fraktálu.

Součástí práce je také základní seznámení se způsoby paralelního programování v dnešní verzi .NET oproti předchůdcům.

2.2 Metodika

Metodika práce je založená na praktickém zpracování aplikace obsahující algoritmus spustitelný sériově i paralelně a následném porovnání vhodnosti a výkonosti těchto variant.

Nejprve autor nastuduje materiály a vypracuje literární rešerši. Na základě získaných informací vyvine sériovou aplikaci pro generaci fraktálu Mandelbrotovy množiny. K sériovému algoritmu pak implementuje paralelní algoritmus tak (podobně), aby mělo smysl je časově porovnávat. Autor pak bude spouštět algoritmus a testovat, aby získal dostatečné množství relevantních dat. Výsledky budou interpretovány.

Pro vývoj bude použit Microsoft C# 6.0 ~ .NET Framework 4.6 a jeho Paralel Task Library.

3 Teoretická východiska

3.1 Jazyk C# a .NET

3.1.1 .NET

Operační systémy Windows od roku 1992 po Windows Server 2008 mají ve svém jádru stejné rozhraní Windows API. S každou novou verzí systému přibývalo v rozhraní API spoustu nových funkcí, ale jednalo se o vývoj a rozšiřování rozhraní, nikoli o jeho nahrazení.

Z důvodu udržování zpětné kompatibility, která je nepostradatelnou vlastností technologií Windows a která je důležitou součástí této platformy, se API stalo nemoderním a nevhodným pro uspokojování požadavků podpory nejnovějšího hardwaru.

Zjednodušeně řečeno je .NET platforma (aplikační rozhraní, API) k programování pro Windows. Spolu s platformou .NET Framework se objevuje jazyk C#, který byl od základů navržen tak, aby spolupracoval s technologií .NET a zároveň využil veškerý pokrok ve vývojových prostředích a koncepcích objektově orientovaného programování, ke kterému došlo během posledních dvaceti let. (Nagel, a další, 2009)

3.1.2 C#

C# je elegantní, typově bezpečný (v každém datovém typu může být uložena jen hodnota odpovídajícího datového typu) a objektově orientovaný jazyk, který dovoluje vývojářům tvořit širokou škálu bezpečných aplikací, které běží na .NET frameworku.

Objektově orientovaným jazykem se myslí podpora základních konceptů objektového programování, mezi něž patří zapouzdření, dědičnost a polymorfismus. ("Intro to the C# Language and the .NET Framework", 2015)

Pointa zapouzdření je v tom, že program využívající nějakou třídu se nemá nijak starat o to, jak třída ve skutečnosti interně funguje. Program prostě vytvoří instanci této třídy a poté volá její metody. Dokud budou metody provádět, co provádět mají, nemá se program starat o to, jak jsou interně naimplementovány. Druhým účelem zapouzdření tříd je skrytí vnitřních informací nutných k běhu metod před programem. (Sharp, 2010)

Dědičnost dovoluje vytvořit hierarchický vztah mezi třídami, který umožní využití funkcionality jedné třídy v jiné. Rozlišujeme tři typy dědičnosti:

- jednoduchá - odvozená třída dědí charakteristiky třídy předka
- vícenásobná - odvozená třída může mít neomezený počet mateřských tříd, přijímá jejich charakteristiky a může je dále rozšiřovat
- úrovňová - je variantou jednoduché dědičnosti, kdy třída je zároveň potomkem nějaké třídy a zároveň rodičem jiné (Hanák, 2009)

Polymorfismus umožňuje objektům volání jedné metody se stejným jménem, ale s jinou implementací.

3.1.3 Lambda výrazy

V práci autor často využívá tzv. „lambda expressions“. Lambda výraz je anonymní funkce, pomocí které lze vytvořit delegát. Použitím lambda výrazu jde vytvořit lokální funkce, které mohou být poslány jako argumenty nebo vráceny jako návratová hodnota funkcí.

K vytvoření delegátu pomocí lambda výrazu se používá operátor „=>“.
("Lambda Expressions (C# Programming Guide)", 2015) (Avidar, 2013) ("Lambda Expressions in PLINQ and TPL", 2010) (Stellman, a další, 2013)

```
delegát:                          (vstupní parametry) => výraz

delegát na anonymní funkci:       (x, y) => x == y
                                  (int x, string s) => s.Length > x
```

```
delegát na metodu mojeMetoda:     () => MojeMethoda()
```

3.1.4 Zámek prostředků Lock

Klíčové slovo lock označí kritickou oblast a zajistí přístup do této oblasti pouze jednomu podprocesu najednou. Pokud se jiný podproces pokusí přistoupit kritickou oblast, bude čekat, dokud se objekt zámek neuvolní. ("lock Statement (C# Reference)", 2015)

Globální proměnné:

```
private Object lockObject = new Object();
       //zamek urceny pro pristup k doublebufferu z vice vlaken najednou
private bool needLocks = false;
       //nastavim na TRUE po dobu behu vice vlaken, abych pouzival zamek
```

Implementace zámku:

```
if (needLocks == false)
       (zapiš bod do bufferu)
else if (needLocks == true)
       lock (lockObject)
              (zapis bod do bufferu)
```

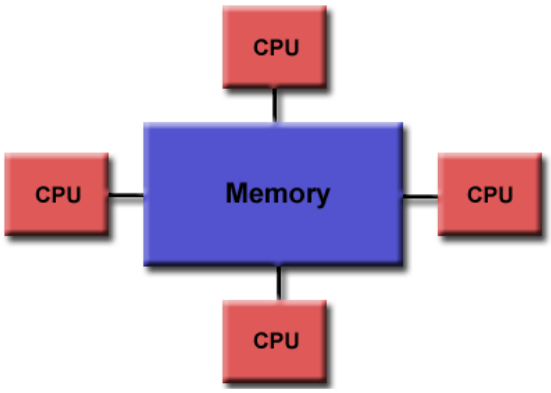
3.2 Paralelismus

Paralelní programování je rozšíření sekvenčního programování a je hlavním způsobem, jak zpracovávat každodenní proud informací. Jeho cílem je sestrojit ty nejrychlejší programy pro paralelní počítače. Proces vývoje se zaměřuje na algoritmus, jazyk a způsob, jakým se program rozvrhne do paralelního počítače. Paralelní algoritmus je dán dílčími sekvenčními algoritmy a vzorem (způsobem), jak je spojit. (Ranjan, 2008)

3.2.1 **Architektury paralelismu**

3.2.1.1 Sdílená paměť

Všechny procesory sdílí veškerou paměť. Mohou tak nezávisle na sobě zpracovávat sdílená data na této paměti uložená.

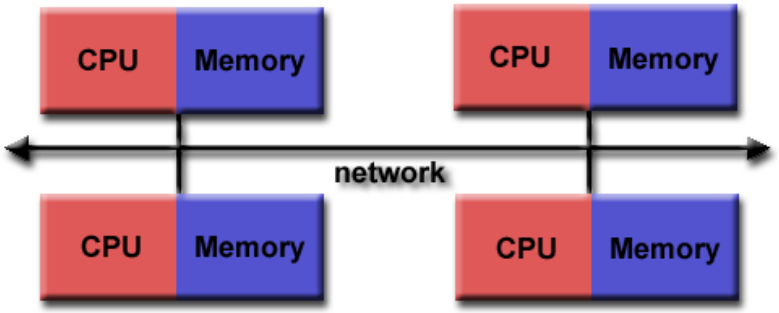


Obrázek 1 Sdílená paměť [1]

3.2.1.2 Distribuovaná paměť

Procesory mají svojí vlastní lokální paměť. Systém potřebuje síťovou komunikaci, aby mohl propojit informace z vnitřních pamětí procesorů.

Procesory pracují nezávisle, a když úloha potřebuje data z paměti náležící jinému procesoru, je na programátorovi, aby zajistil komunikaci a synchronizaci. (Blaise, 2016)

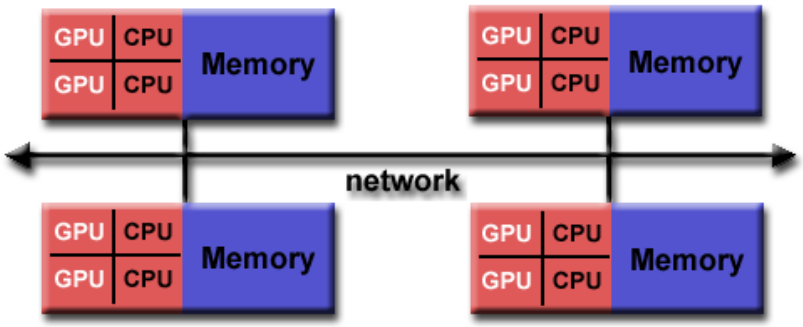


Obrázek 2 Distribuovaná paměť [2]

[1] Zdroj: https://computing.llnl.gov/tutorials/parallel_comp/
[2] Zdroj: https://computing.llnl.gov/tutorials/parallel_comp/

3.2.1.3 Hybridní: distribuovaná – sdílená paměť

Skupina centrálních/grafických procesorových jednotek sdílí paměť a je síťově propojena s dalšími takovými skupinami, jako to bylo v distribuované architektuře. (Blaise, 2016)



Obrázek 3 Hybridní paměť[3]

3.2.2 **Modely paralelismu**

3.2.2.1 Model sdílené paměti (bez použití vláken)

Asynchronní procesy sdílí společný adresní prostor, ke kterému přistupují pomocí zámků/semaforů

3.2.2.2 Paralelní model dat

Více *tasků* má zpracovat data uložené na sdíleném poli. Nejprve proběhne rozdělení *adresního prostoru* tak, že každý *task* má přístup jen do své části pole. Následně mohou přistupovat k datům souběžně.

3.2.2.3 Model threadů (vláken)

Rozdělení procesu na jednotlivá vlákna, která mohou běžet souběžně.
(Blaise, 2016)

[3] https://computing.llnl.gov/tutorials/parallel_comp/

3.2.2.4 .NET 4.0 Task Parallel Library

Uvnitř Task Parallel Library (TPL) byla zapouzdřena celá správa threadů (vláken), tj. starost o vytváření, ukončování a správu jejich množství závisle podle množství dostupných procesorů. ("What's New in the .NET Framework 4", 2012)

3.2.2.4.1 System.Threading.Tasks.Task.Factory

Spouštění Action uvnitř asynchronních tasků.

3.2.2.4.2 Parallel.Invoke

Spustí pole delegátů paralelně a počká, než se všechny dokončí.

("Task-based Asynchronous Programming", 2012)

3.2.2.4.3 Parallel.For

Paralelní ekvivalent for cyklu v C#. Počká na všechny iterace.

3.2.2.4.4 Parallel.ForEach

Paralelní ekvivalent foreach smyčky. Spustí všechny prvky dané kolekce a počká na dokončení všech položek.

```
foreach (var i in collection)
{
	DoSomeWork(i);
}

// Parallel.ForEach(IEnumerable<TSource> source, Action<TSource> body)
Parallel.ForEach(collection,
	(item) =>
	{
		DoSomeWork(item)
	});
```

("How to: Write a Simple Parallel.ForEach Loop", 2012)

3.2.2.5 .NET 4.5 Příkazy Async a Await

Možnost zapouzdřit metodu pomocí třídy Task za účelem jejího asynchronního spuštění **uvnitř synchronního kódu**. Při nárazu na klíčové slovo await vrátí řízení mateřskému prvku. Po dokončení asynchronní části se pokračuje následujícím příkazem jako v klasickém synchronním programování. ("Asynchronous Programming with async and await C#", 2015)

Pokud na kliknutí tlačítka naimplementujeme metodu, která zabírá nějakou dobu (například stahování z webu WebClient.DownloadString), akce provedeny na UI budou zpožděny, což má za následek zamrzlé UI. Ačkoli pokud použijeme asynchronní verzi metody (příkazy async / await), požadavek okamžitě odevzdá řízení. (Bray, 2012)

Task.Factory.StartNew modernizováno na Task.Run: (Toub, 2011)

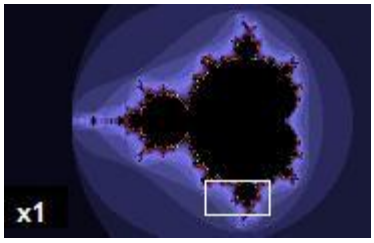
```
Task.Factory.StartNew(someAction, CancellationToken.None,
       TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);

// identický účinek příkazů

Task.Run(someAction);
```

Ukázka implementace async/await:

```
// vlastní příklad implementace volání await dvojitým zapouzdřením metody
private void button1_Click(object sender, EventArgs e)
{
       CallLongOperationAsync();
}
private async void CallLongOperationAsync()
{
       await LongOperationTask();
}
private Task LongOperationTask()
{
       return Task.Run(
              () => LongOperation()
       );
}
private void LongOperation()
{
       // moje dlouha operace
}
```

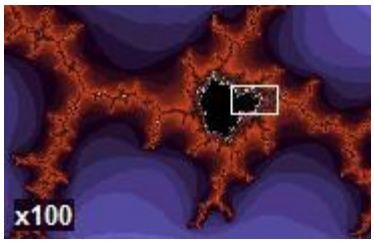
3.3 Fraktály

Fraktál (z latinského *fractus* - rozbitý) je geometrický objekt, který je „soběpodobný“ (lze vypozorovat opakující se určitý motiv v jakémkoli měřítku přiblížení), má velmi složitý tvar a je generován pomocí jednoduchých pravidel.

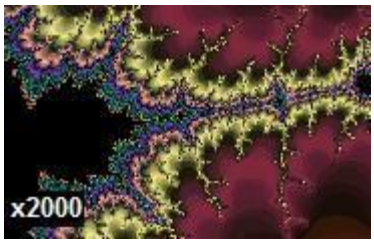


Obrázek 4 Mandelbrot přiblížení 1x [4]

Obrázek 5 Mandelbrot přiblížení 6x [4]



Obrázek 6 Mandelbrot přiblížení 100x [4]



Obrázek 7 Mandelbrot přiblížení 2000x [4]

I přesto, že je tvořen poměrně jednoduchým matematickým algoritmem, jsou to jedny z nejsložitějších geometrických objektů, které matematika zkoumá.

[4] Všechny 4 obrázky: Zdroj: https://cs.wikipedia.org/wiki/Frakt%C3%A1l

3.3.1 **Mandelbrotova množina**

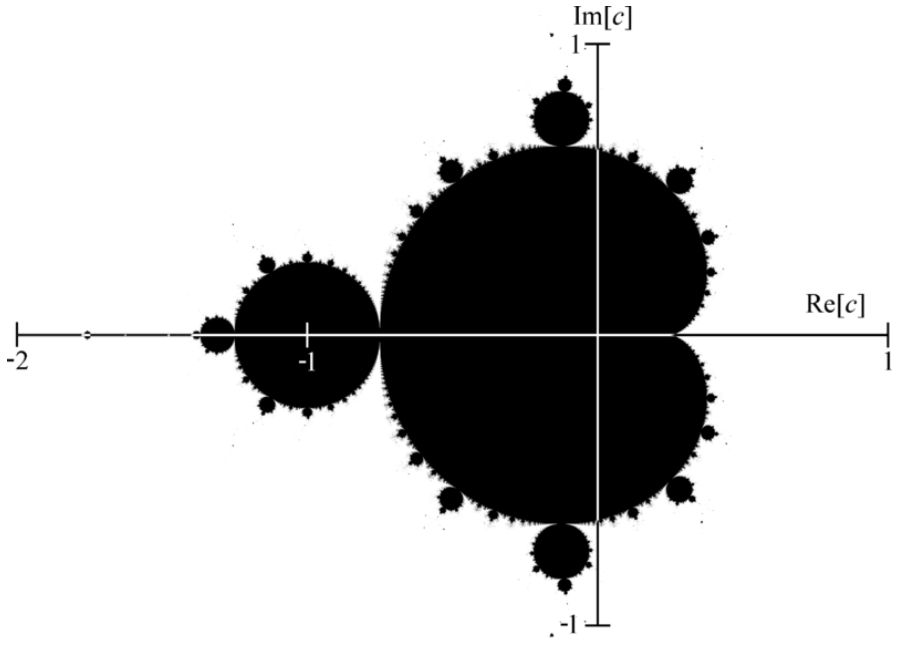
3.3.1.1 Definice

Mandelbrotova množina je množina bodů komplexní roviny, získaných z kvadratické rekurzivní rovnice:

$$z_{n+1} = z_n^2 + C, \qquad \text{kde} \quad z_0 = 0$$

Množina je definována jako množina komplexních čísel C, pro která absolutní hodnota členů posloupnosti z_n nepřekročí hodnotu 2 (posloupnost je omezená).

Shishikura (1994) dokázal, že hranice Mandelbrotovy množiny je Fraktál s Hausdorffovou dimenzí 2. (Weisstein, 2009)



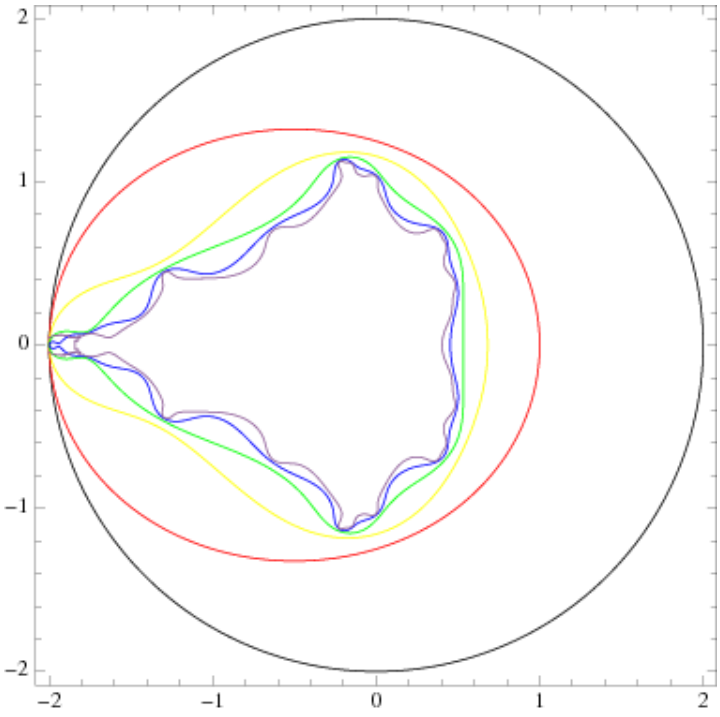
Obrázek 8 Umístění mandelbrot setu v imaginární rovině [5]

[5] Zdroj: http://www.dialogues-cns.com/publication/a-history-of-chaos-theory/

3.3.2 **Vizualizace**

Pro grafické znázornění Mandelbrotovy množiny se limit, který určuje, že body unikly, aproximuje na číslo *iterationsMax* místo ∞.

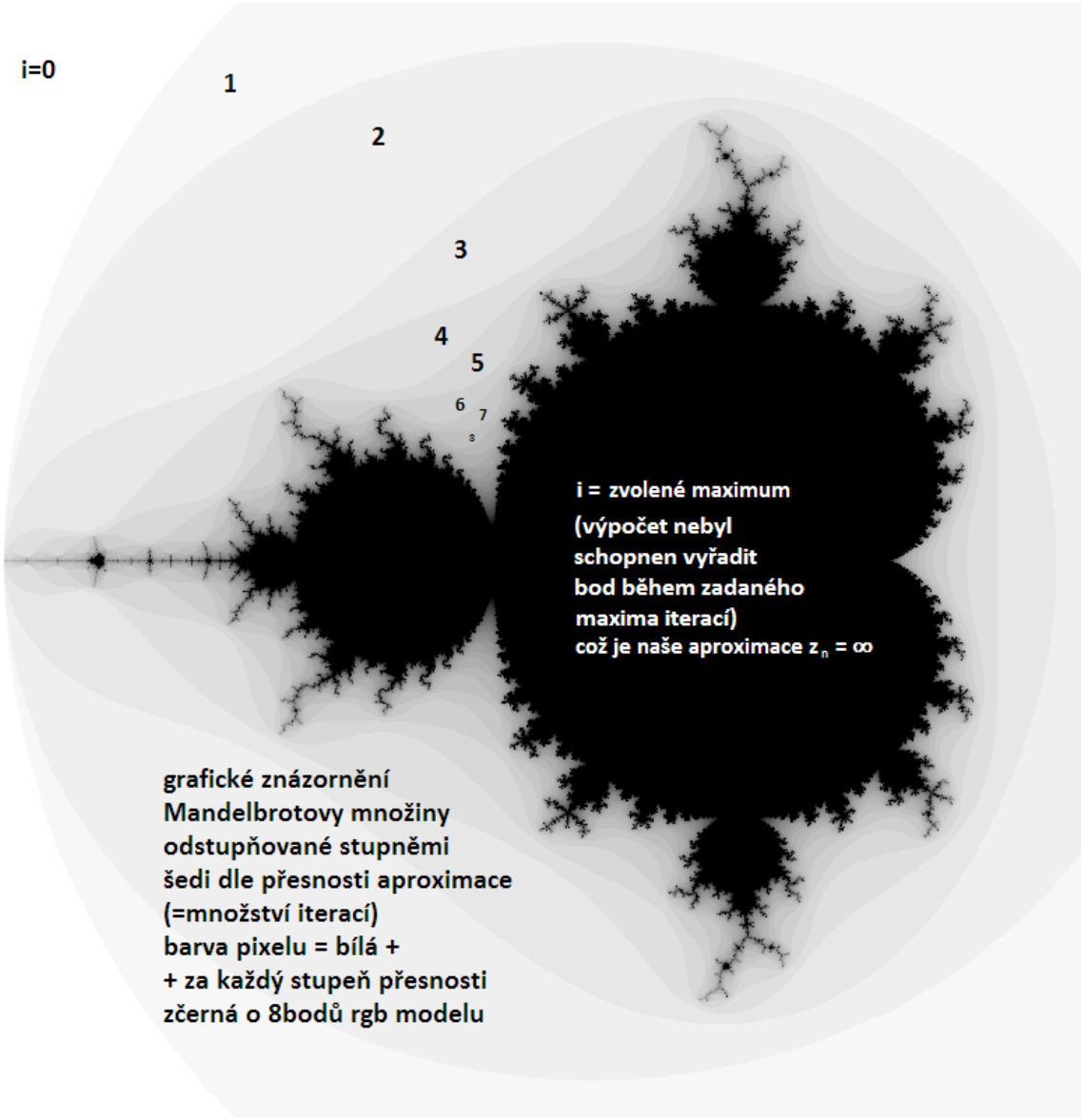
Překrásné počítačově generované grafy pak jdou vypracovat vybarvením čísel **nepatřícím** do množiny barvou na základě toho, na kolikáté iteraci jsme zjistili, že přestaly patřit do Mandelbrotovy množiny (viz obrázek níže). (Weisstein, 2009)



Obrázek 9 Vizualizace Mandelbrotu s maximem 6 iterací [6]

[6] Zdroj: http://mathworld.wolfram.com/MandelbrotSet.html

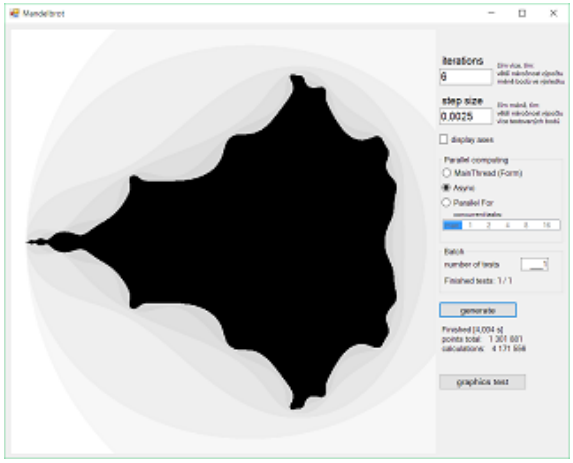
Následující 4 obrázky ilustrují způsob vizualizace:



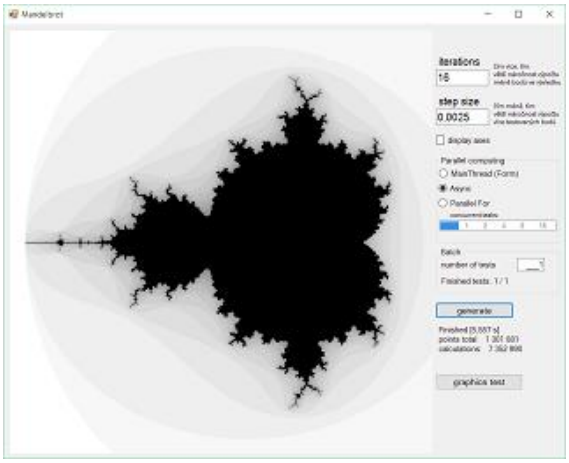
Obrázek 10 Očíslované stupně úniku bodů[7]

[7] Vlastní zdroj.

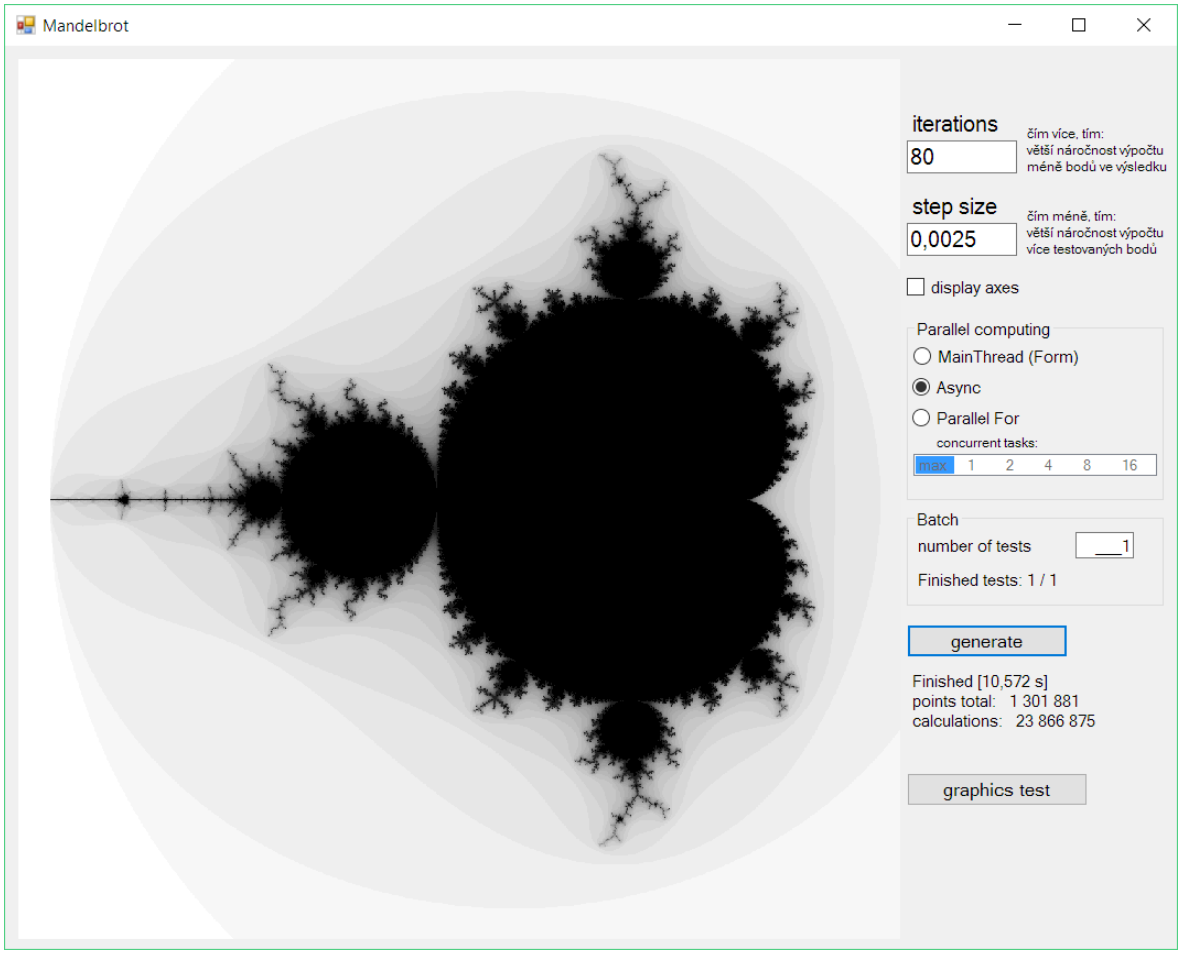
Obrázky z praktické implementace znázorňující postupné pokračování aproximace.



Obrázek 11 Mandelbrot i=6



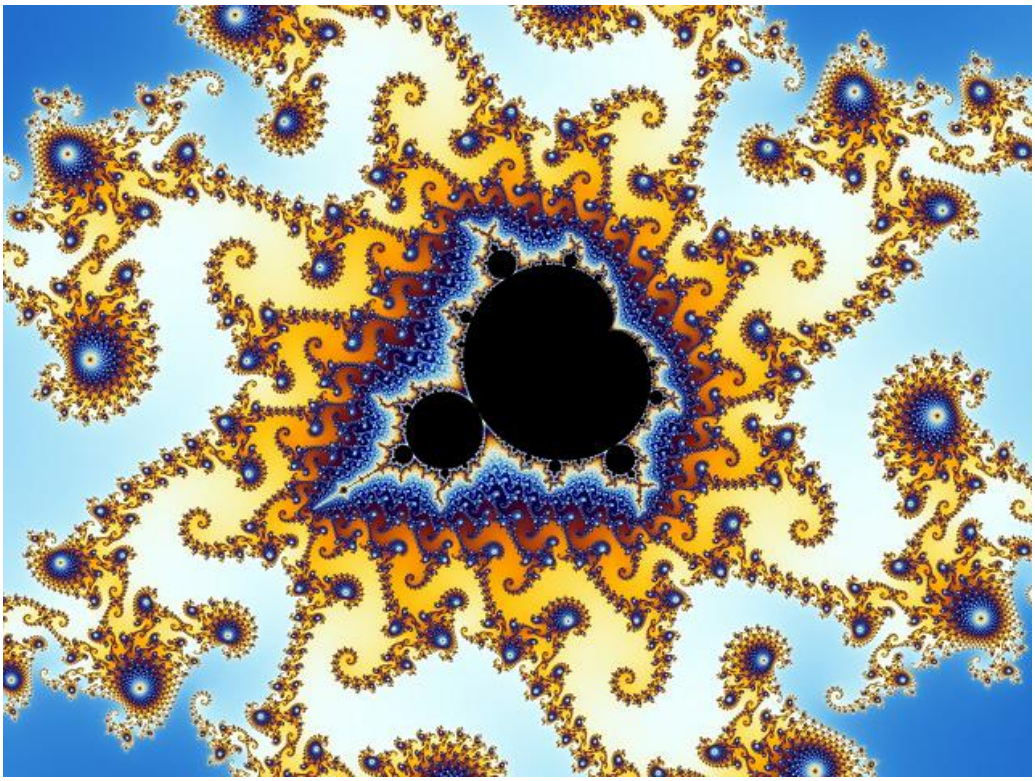
Obrázek 12 Mandelbrot i=16



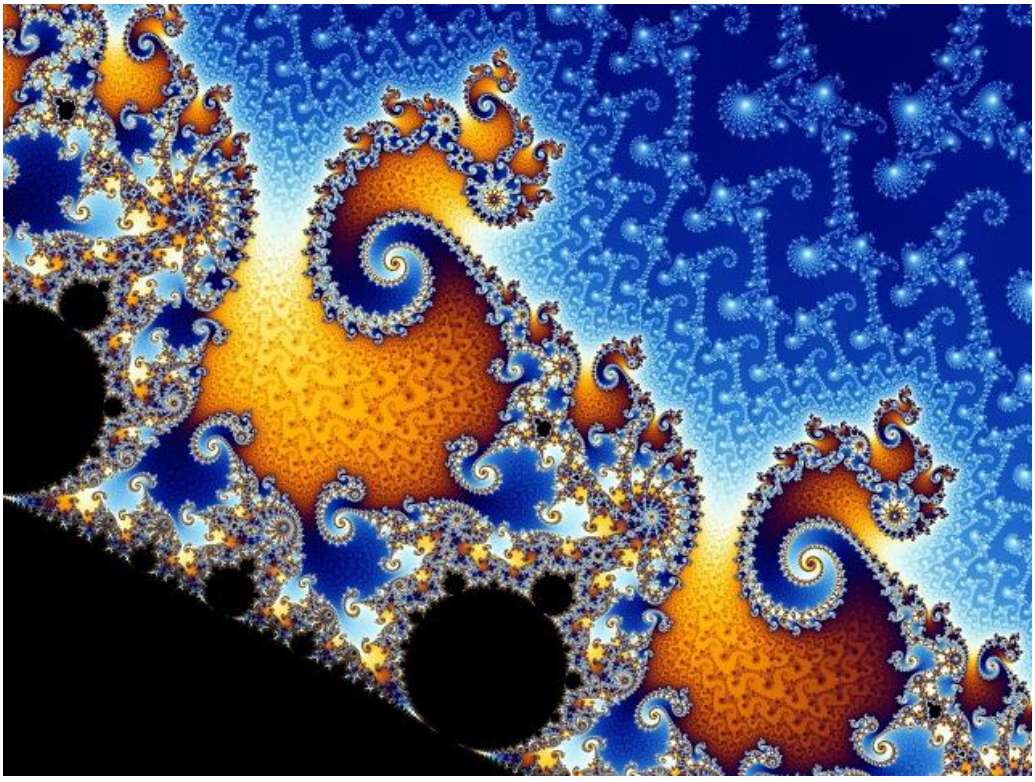
Obrázek 13 Mandelbrot i=80 [8]

[8] Vlastní zdroj.

3.3.3 Ilustrační ukázky s profesionální grafikou (z webu)



Obrázek 14 Ukázka profesionální vizualizace[9]



Obrázek 15 Ukázka profesionální vizualizace 2[10]

[9] Zdroj: https://science-et-fiction.fr/sciences/physique-quantique/
[10] Zdroj: http://www.mathemania.com/the-mandelbrot-set/

4 Vlastní práce

4.1 Východiska

4.1.1 Předpoklad

Autor chtěl ověřit předpoklad, že využitím více jader/vláken k řešení problému dojde ke snížení času, který je požadován na vyřešení tohoto problému.

To ovšem není univerzální pravda, zaprvé ne každou práci lze efektivně rozdělit pro jednotlivá vlákna. Úkoly, které by mohly být rozdány jednotlivým vláknům, musí mít tu vlastnost, že na sobě nejsou sekvenčně závislé, aby mohly skutečně běžet souběžně (popřípadě jsou na sobě závislé jen částečně a je mezi nimi vyřešena synchronizace informací).

Zadruhé, časový přínos kódu spuštěného paralelně musí být větší než (časová) cena na jeho vytvoření a správu.

4.1.2 Výběr algoritmu

Aby bylo možné změřit a porovnat několik kombinací těchto kritérií, byl vybrán matematický algoritmus, který umožňuje zvolit parametry náročnosti paralelních částí kódu, stejně tak jako množství požadovaných výpočtů v každém testu. Pro tento účel autor vybral proces generování jednoho z nejznámějších fraktálů, Mandelbrotovu množinu (Mandelbrot set), který kromě volitelné matematické komplexity oplývá i svou úžasnou elegancí.

Autor se rozhodl, že nejprve vytvoří jádro programu generující Mandelbrotovu množinu, kterou bude následně zapouzdřovat pro potřeby paralelizace.

Později se ukázalo, že naprogramovat Mandelbrot set tak, aby tělo kódu bylo shodné a použitelné pro sekvenční i paralelní spouštění, je vyzývavým úkolem.

4.2 UI - Formulář aplikace

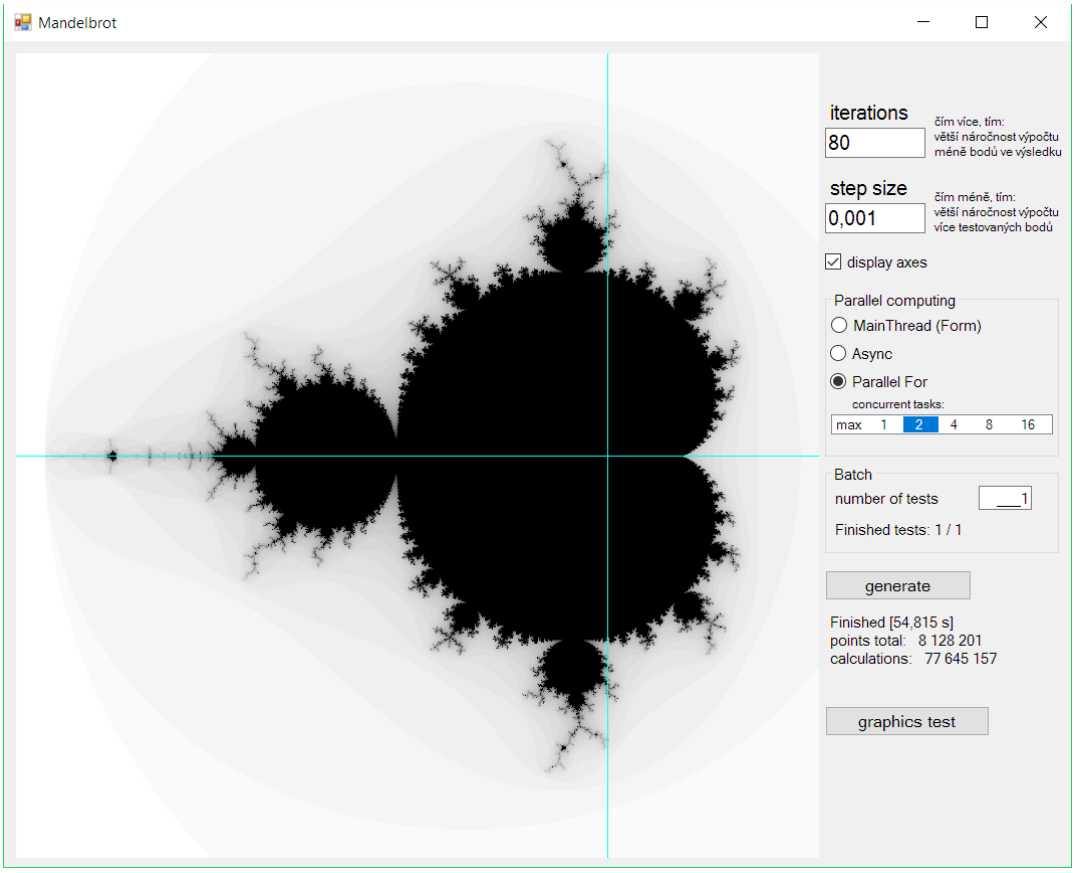
4.2.1 Objekt Graphics – renderovací plátno

Nejprve byla rozhodnuta velikost plátna, na které se bude renderovat spočítaný fraktál.

Toto plátno bylo realizováno jako objekt Graphics vykreslovaný na PictureBox s dimenzemi 800x800 pixelů.

V práci se o ploše určené k renderování pro zjednodušení mluví také jako o plátně či PictureBoxu, i když to není úplně přesné. Popsáno v kapitole 4.3.

Velikost formuláře aplikace autor zvolil tak, aby se do něj plátno a další ovládací prvky pohodlně vešly.



Obrázek 16 UI Formuláře[11]

Protože komplexnější vykreslování a zoom není záměrem práce, FormBorderStyle bylo nastaveno na FixedSingle, proto není potřeba řešit změnu velikosti prvků.

[11] Vlastní zdroj.

4.2.1.1 Velikost zkoumané plochy pro fraktál

Jak již bylo vysvětleno v kapitole 3.3.1, všechny body Mandelbrotovy množiny leží do vzdálenosti 2 od středového bodu.

To by znamenalo spočítat body uvnitř rozsahu od [-2,-2] po [2,2].

Autor zvolil, že vhodné přiblížení (rozsah výpočtů) bude [-2,1;-1.3] až [0,75;1,3].

```
//rozsah, ve kterem ma vyznam pocitat body mandelbrot setu
private double _xMin = -2.1, _xMax = 0.75, _yMin = -1.3, _yMax = 1.3;
```

4.2.1.2 Přizpůsobení poměru stran polygonu bufferu

Protože PictureBox, na který se bude renderovat výsledek výpočtů je čtverec a vymezené rozmezí (viz 4.2.3) je obdélník, v závislosti na tvaru obdélníku by na plátně chyběla vypočítaná plocha, nebo naopak přebývala plocha taková, kterou není kam vykreslit.

Autor se rozhodl, že zanechá hodnoty _xMin a _xMax. Rozmezí y je upraveno funkcí adjustSize.

```
#region Common methods
private double _scaleX;
private void adjustSize() //nastavi pomer roztahnuti desetinnych souradnic na
pixely podle sirky platna + roztahne vertikalni souradnice tak, aby vypocty
pokryly cele platno
{
	_scaleX = pictureBox.Width / (_xMax - _xMin);
	//scale pro Y nechci, protoze chci zachovat pomer stran. budu se tedy
	ridit stranou x

	//abych nemel v pictureboxu cerne pruhy nahore a dole, budu pocitat i
	tyto body = rozsirim Ymin/Ymax
	double rangeY = pictureBox.Height / _scaleX;
	_yMin = -0.5 * rangeY; //-1,425 pri ctvercovem rozmeru 800x800 platna
	_yMax = +0.5 * rangeY; //+1,425
}
```

_scaleX je poměr, kolik bodů na renderovatelném plátně znázorňuje jeden bod na souřadnicích komplexních čísel. Vychází, že [1,0] je ~280,7 pixelů od [0,0].

4.2.2 Vstupní parametry a jejich validace

Prvními ovládacími prvky jsou TextBox nadepsaný „iterations“ a TextBox nadepsaný „step size“.

Hodnota „iterations“ je maximální množství čísel posloupnosti, které se budou v každém bodě kontrolovat, viz *iterationsMax* v kapitole 3.3.2. Logicky, čím více členů posloupnosti se bude u každého bodu kontrolovat, tím více vzroste výpočetní náročnost a zároveň se zvýší detaily vypočítaného fraktálu.

Hodnota „step size“ určuje vzdálenost bodů komplexní roviny od sebe navzájem. Čím menší vzdálenost, tím více bodů, tím více výpočetních operací, tím větší spočítané rozlišení fraktálu.

Oba tyto prvky jsou chráněny proti zadání nekorektní hodnoty.

```
private void txtIterations_Validating(object sender, CancelEventArgs e)
{
	int i;
	TextBox txtBx = sender as TextBox;
	if (!Int32.TryParse(txtBx.Text, out i))
	{
		txtBx.BackColor = Color.Red;
		e.Cancel = true;
	}
	Else	txtBx.BackColor = Color.White;
}
```

```
private void txtStepSize_Validating(object sender, CancelEventArgs e)
{
	double i;
	TextBox txtBx = sender as TextBox;
	if (!Double.TryParse(txtBx.Text, out i))
	{
		txtBx.BackColor = Color.Red;
		e.Cancel = true;
	}
	Else	txtBx.BackColor = Color.White;
}
```

4.2.3 Osy a transformace souřadnic

CheckBoxem „display axes“ si lze vybrat, zdali se po vygenerování fraktálu znázorní osa. Na demonstračním obrázku je vidět tyrkysovou barvou.

Pochopitelně se jedná o osu v souřadnicovém systému fraktálu (souřadnice [0,0] je téměř ve středu), ne o index pixelu v PictureBoxu (kde souřadnice [0,0] je levý horní pixel.)

```
if (checkBxDisplayAxes.Checked)
{
   //horizontalni osa
   gfxB.Graphics.DrawLine(Pens.Cyan,
      (float)((-_xMin) / (_xMax - _xMin)) * pictureBox.Width, 0,
      (float)((-_xMin) / (_xMax - _xMin)) * pictureBox.Width,
       pictureBox.Height);
   //vertikalni osa
   gfxB.Graphics.DrawLine(Pens.Cyan,
      0, pictureBox.Height / 2,
      pictureBox.Width, pictureBox.Height / 2);
}
```

Horizontální osa je jen zasazená do středu plátna, vertikální osa je však posunutá tak, aby rozsah hodnot x (_xMin = -2.1, _xMax = 0.75) vyplnil celý prvek. Osa tedy bude v ~73,7% šířky prvku PictureBox, protože -(-2,1)/(-0,75-2,1) = 0,7368421…

4.2.3.1 Transformace souřadnic zapisovaného bodu

Jak již bylo předvedeno v předešlém odstavci, z důvodu toho, že prvek do kterého je kresleno, má střed souřadnic [0,0] jinde, než kde autor chce střed souřadnic mít, musí každý bod při jeho zápisu na PictureBox transformovat.

Každý spočítaný bod Mandelbrotovy množiny ve formátu komplexního čísla je transformován na bod na plátně tak, že je vynásoben koeficientem _scaleX a sečten s korespondující osou.

_scaleX je poměr, kolik bodů na renderovatelném plátně znázorňuje jeden bod na souřadnicích komplexních čísel. Vychází, že [1,0] je ~280,7 pixelů od [0,0].

```
double _scaleX = pictureBox.Width / (_xMax - _xMin);
x na plátně = (x * _scaleX + (((-_xMin) / (_xMax - _xMin)) * pictureBox.Width))
y na plátně = (y * _scaleX + (0.5 * pictureBox.Height))
```

4.2.4 Úroveň paralelismu

V uživatelském rozhraní je dále skupina RadioButtonů, pomocí kterých je možné si vybrat úroveň paralelismu.

Vybráním možnosti parallel se odemkne možnost vybírat v ListBoxu maximální povolené množství souběžných běžících Tasků.

```
private void radioParallelFor_CheckedChanged(object sender, EventArgs e)
{
      if (radioParallelFor.Checked) listBoxConcurrency.Enabled = true;
      else                          listBoxConcurrency.Enabled = false;
}
```

Naplnění ListBoxu s množstvím spouštených tasků uvnitř Form1_Load metody:

```
listBoxConcurrency.DataSource = new BindingSource(
new Dictionary<int, string>() { { -1, " max" }, { 1, "    1" },
{ 2, "    2" }, { 4, "    4" }, { 8, "    8" }, { 16, "    16" } }, null);
```

4.2.5 **Testování po dávkách, dynamické labely, tlačítka**

MaskedTextBox nadepsaný number of tests slouží k opakování celého testu vícekrát za účelem statistiky a díky masce „###0“ do sebe dovoluje zapsat jen jedno až čtyřmístný int. Přidružený Label *labTestCount* pak během běhu označuje, kolik testů je již hotovo a kolik ještě zbývá.

Pod tlačítkem „generate“ spouštějící výpočet fraktálu je Label *labRunStatus*, který za běhu výpočtů udává množství již spočítaných bodů a po dokončení výpočtu udá i čistý čas, který zabral výpočet Mandelbrotovy množiny.

Množství spočítaných bodů (points, pointsTotal, points done) neznamená množství zobrazených pixelů. Protože máme fixní plátno o 640000 pixelech (800x800), množství bodů může tedy být, podle parametrů zadání, menší či větší.

Méně bodů spočítaných než pixelů na plátně má za následek mezery mezi vyrenderovanými body.

Více bodů spočítaných než pixelů na plátně má za následek, že se body, jež po transformaci dostanou přidělený stejný bod plátna, přepíšou novějším. To ale nevadí, protože se jedná jen o grafické znázornění velmi drobných detailů, které je spíše bonusovou funkcí aplikace.

V uživatelském prostředí se také nachází tlačítko „graphics test“, který sloužilo pro účely hledání a odstraňování chyb, testování a dalšího debugu v průběhu celého vývoje aplikace. Pro release by asi bylo vhodnější tuto ladící funkci skrýt.

4.3 Implementace renderované plochy – „plátna“

Autor usoudil, že vykreslovat každý bod rovnou na UI by zabralo moc času a tím i ovlivňovalo dobu výpočtů výsledků. Proto se rozhodl, že využije grafického bufferu v paměti, který jednou za čas nechá vyrenderovat, aby uživatel viděl postup. K tomuto účelu slouží třída System.Drawing.BufferedGraphics.

Nejprve byly deklarovány dva globální private objekty z Graphics a BufferedGraphics a vytvořena grafika na PictureBoxu

```
private Graphics gfx;
private BufferedGraphics gfxB;
private void Form1_Load(object sender, EventArgs e)
{
        //zavedeni buffered grafiky do ktere pak mohou metody kreslit
        gfx = pictureBox.CreateGraphics();
```

Předtím, než je možné začít počítat fraktál a tedy i zapisovat do bufferu je nutné ho alokovat. Toto je děláno uvnitř každého testu (výpočet jednoho celého fraktálu) zvlášť a na konci testu je volána funkce Dispose() kvůli uvolnění prostředků z paměti.

```
BufferedGraphicsContext bufferedGContext = BufferedGraphicsManager.Current;
gfxB = bufferedGContext.Allocate(gfx, this.DisplayRectangle);

        (…) – tělo výpočtů
        (…) – zápis/kreslení do gfxB

gfxB.Render();
gfxB.Dispose();
```

4.4 Implementace výpočtu Mandelbrotovy Množiny (metoda)

Autor se snažil vytvořit takový kód, který půjde spustit pro všechny měřené varianty paralelismu bez, či jen s minimálními změnami, aby se tyto rozdíly neprojevily na měření.

Základní algoritmus „Escape time algorithm“ získání Mandelbrotovy množiny je zvolení si množství (hustoty) bodů v dvourozměrném poli a pro každý bod spočítat, zda právě tento bod do množiny patří či nikoliv, následně vybarvit body způsobem popsaným v kapitole 3.3.2.

Běžnou praxí procházení dvourozměrných polí při programování je využití integerů, tedy celých čísel, jako indexů pro osy X a Y. V tomto případě se však autor rozhodl, že je kvůli podstatě hodnot používaných pro výpočet bodů Mandelbrotovy množiny vhodné nepřepočítávat indexy z desetinných čísel o vysoké přesnosti na celá čísla a nechat je uložené v datovém typu double. Toto rozhodnutí přispělo i k větší přehlednosti a konzistenci informací ve všech částech kódu.

4.4.1 Inicializace proměnných

```
#region SERIAL
private void MandelbrotMainThreadSync() //calculate Mandelbrot set in form's
MAIN thread
{
	//nasledujici promenne je nutno nastavit pred smyckou for, aby byl cely
	soubor mereni stejny
	_iterationsMax = Convert.ToInt32(txtIterations.Text);
	_stepSize = Convert.ToDouble(txtStepSize.Text);
```

V operacích výše nemůže dojít k chybě, protože validace čísla již proběhla na eventu validating.

```
	//_stepSize = Convert.ToDouble(txtStepSize.Text,
	System.Globalization.NumberFormatInfo.InvariantInfo);
	//ignoruje formatovani lokalizace, takze ocekava desetinnou TECKU
```

Zajímavý poznatek je, že funkce Convert je citlivá na kulturu OS a podle ní požaduje formátování (např. desetinná tečka v anglicky mluvících zemích, u nás čárka).

Toto chování lze vypnout, viz zakomentovaný kód výše.

```
	adjustSize(); //ziskani pomeru _scaleX pro prevadeni souradnic na pixely
```

AdjustSize (v kódu výše, implementace v kapitole 4.2.1.2) se volá zaprvé k získání poměru pro transformaci bodů, zadruhé k upravení yMin, yMax hodnot, aby poměr plátna 1:1 souhlasil s velikostí čtverce na komplexní rovině.

```
testCountTotal = Convert.ToInt32(maskedTxtBxTestCount.Text);
//nemuze hodit exception, pole muze obsahovat jen int
```

Získání počtu opakování celého výpočtu Mandelbrot setu neboli velikosti dávky testů (viz výše).

4.4.2 Možnost vícenásobného opakování - dávka

Celý kód, kromě výše zmíněné inicializace proměnných (kapitola 4.4.1), byl zabalen do for cyklu, je tedy vykonán pro každý test z dávky.

Všechny následující kapitoly (4.4.3 až 4.4.9) se odehrávají uvnitř následujícího cyklu.

```
for (testCountCompleted = 0;
      testCountCompleted < testCountTotal; testCountCompleted++)
```

4.4.3 Základní smyčka For

```
for (double i = _xMin; i <= _xMax; i += _stepSize)
{
      for (double j = _yMin; j <= _yMax; j += _stepSize)
      {
            (spočítání dle parametrů, zda bod [i,j] patří do Mandelbrot setu
            a zapsání výsledku do bufferu (do transformovaných souřadnic))
      }
}
```

4.4.4 Jádro aplikace – vnitřek základního for cyklu

4.4.4.1 Sériový návrh

```
for (double i = _xMin; i <= _xMax; i += _stepSize)
{
	for (double j = _yMin; j <= _yMax; j += _stepSize)
	{
		calculationsTotal += DoesDiverge(i, j); //draw funkce je uvnitr
	}
}
```

Funkce DoesDiverge vrací množství provedených iterací každého bodu. Tato informace slouží pouze pro zajímavost.

Využití struktury System.Numerics.Complex poskytlo autorovi přehlednou a elegantní práci s komplexními hodnotami, jež vstupují do vzorce ke kalkulaci bodů Mandelbrotovy množiny.

Základní for loop spustí pro každý bod kontrolu, zdali patří do Mandelbrotovy množiny a následně spustí fukci Draw, jenž bod zapíše do bufferu s příslušnou barvou.

To, v které iteraci byl daný bod vyřazen z množiny, je odesláno funkci Draw jako třetí parametr (-1 v případě, že v zadaném maximálním množství iterací vyřazen nebyl).

```
//---------------------------------------------------------------------
private long DoesDiverge(double x, double y) //zda-li cislo po danem mnozstvi
iteraci opusti mandelbrot set
{
	Complex c = new Complex(x, y);
	Complex z = 0;
	int i;
	for (i = 0; i < _iterationsMax; i++)
	{
		z = Complex.Add(Complex.Pow(z, 2), c); //z = z^2 + c
		if (z.Magnitude > 2)
		{//pokud cislo opusti vzdalenost 2 od nuly, nepatri do Man.setu
			Draw(x, y, i);
			return i;
		}
	}
	Draw(x, y, -1); //-1 znamená nekonecno iteraci = pocet iteraci
		dosel na zvoleny max = bod je soucasti mandelbrot setu
	return i;
}
```

4.4.4.2 Asynchroní responzivní návrh

Základní for cyklus není pro využití async/await potřeba nijak přizpůsobovat.

4.4.4.3 Paralelní návrh pomocí Task Library

```
for (double i = _xMin; i <= _xMax; i += _stepSize)
{
	for (double j = _yMin; j <= _yMax; j += _stepSize)
	{
		calculationsTotal += DoesDiverge(i, j); //draw funkce je uvnitr
	}
}
```

Sériový základní cyklus z kapitoly 4.4.4.1 (výše) byl upraven způsobem popsaným v kapitole 3.2.2.4.

Z důvodu, že se autor rozhodl v iteračních krocích používat hodnoty typu *double* (viz 3. odstavec kapitoly 4.4), bylo nutné využít Parallel.ForEach místo Parallel.For, protože Parallel.For podporuje pouze iterátor typu *integer*.

Autor daný problém vyřešil použitím funkce Iterate (Toub, 2009), která vytvoří kolekci hodnot i, kterou předtím procházela smyčka for.

```
//Iterator pro Parallel.ForEach, protoze Parallel.For nemuzu pouzit vlastni
increment i(index) typu double
private static IEnumerable<double> Iterate(
	double fromInclusive, double toExclusive, double step)
{
	for (double d = fromInclusive; d < toExclusive; d += step)
		yield return d;
}
```

Tato kolekce se snadno vloží do prvního parametru Parallel.ForEach (viz kód níže), čímž se dosáhne požadovaného chování.

Dále se čte způsobem popsaným v kapitole 4.4.7.2 hodnota ListBoxu, kde uživatel zadal úroveň paralelizace – množství souběžných tasků. Toto číslo se předá v datovém typu ParallelOptions.MaxDegreeOfParallelism jako parametr pro Parallel.ForEach.

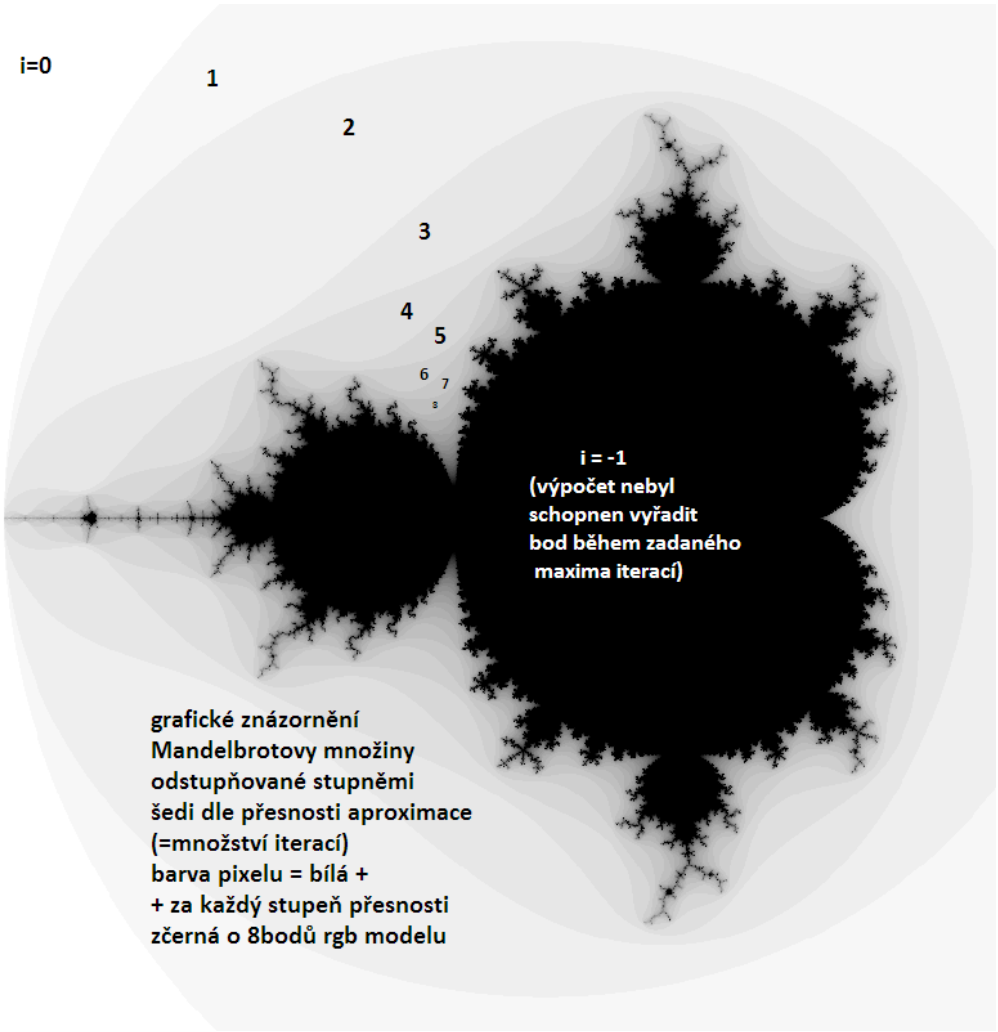
```
ParallelOptions pOptions = new ParallelOptions();
listBoxConcurrency.Invoke(new Action(
       () => { pOptions.MaxDegreeOfParallelism =
Convert.ToInt32(listBoxConcurrency.SelectedValue.ToString()); }
));


Parallel.ForEach(Iterate(_xMin, _xMax, _stepSize), pOptions, (i) =>
{
for (double j = _yMin; j <= _yMax; j += _stepSize)
{
calculationsTotal += DoesDiverge(i, j); //draw funkce je uvnitr
}
//pro paralelni vypocty pricitam mnozstvi bodu ve sloupci za kazdym hotovým sloupcem.
toto cislo pak vypisu
calculaionPoints += Convert.ToInt64(Math.Floor(((_xMax - _xMin) / _stepSize) + 1));
labRunStatus.Invoke(new Action(() =>
{
labRunStatus.Text = "Running: " + calculaionPoints.ToString("000 000 000") + " points
done";
labRunStatus.Update();
}));

lock (lockObject) gfxB.Render(); //vykresleni po kazdem sloupci
});
```

4.4.5 **Funkce Draw**

Autor si vybral, že mu stačí vybarvovat body v odstínech šedé. Význam odstínů bodů je popsán v kapitole 3.3.2.

Autor se na základně toho, že v grafickém znázornění bez zoomu nejsou detaily na úrovni asi tak 30. iterace vidět rozhodl, že od 31. iterace mohou být body nadále vybarveny stejně. Toto rozhodnutí má za následek to, že přestože je možné spočítat fraktál na větší přesnost, graficky bude znázorněn jen na 31 iterací. Zároveň toto číslo rozděluje spektrum šedi mezi bílou a černou na takové množství odstínů, aby bylo možné je lidským okem pohodlně rozeznat.

Obrázek je zkopírován z kapitoly 3.3.2 kvůli názornosti algoritmu.



Obrázek 17 Očíslované stupně úniku bodů 2[12]

[12] Vlastní zdroj.

V první části metoda Draw dojde k výběru barvy pro pixel dle výše zmíněných pravidel. (Pozn.: Barva Color.FromArgb(255,255,255) je bílá, Color.FromArgb(0,0,0) je černá.)

V druhé části proběhne transformace souřadnic bodu (vysvětleno a rozepsáno viz kapitola 4.2.3) a zápis bodu do bufferu v paměti (lock vysvětlen v kapitole 3.1.4).

```
private void Draw(double x, double y, int iterationsCount)
{
	int colorNum = 255 - iterationsCount * 8; //posouvam barvu od bile k cerne o 8 bodu rgb modelu za kazdou iteraci
	// *8 znamena, ze mam presnost na 31 iteraci, dalsi jsou cerne. *4 znamena presnost na 63 iteraci. *6 = 42iteraci
	if (colorNum < 0) colorNum = 0; //pokud "dosla paleta barev", pouzij cernou
	if (iterationsCount == -1) colorNum = 0;  //pouzij cernou pro hodnoty patrici do mandelbrot setu
	if (needLocks == false)
		gfxB.Graphics.FillRectangle(new SolidBrush(
				Color.FromArgb(colorNum, colorNum, colorNum)),
				(float)(x * _scaleX + (((-_xMin) / (_xMax - _xMin))
					* pictureBox.Width)),
				(float)(y * _scaleX + (0.5 * pictureBox.Height)),
				1, 1);
	else if (needLocks == true)
		lock (lockObject)
			gfxB.Graphics.FillRectangle(new SolidBrush(
				Color.FromArgb(colorNum, colorNum, colorNum)),
				(float)(x * _scaleX + (((-_xMin) / (_xMax - _xMin))
					* pictureBox.Width)),
				(float)(y * _scaleX + (0.5 * pictureBox.Height)),
				1, 1);}
```

4.4.6 Diagnostický nástroj na měření času

```
System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
sw.Start();
	(…základní smyčka For…)
sw.Stop();
```

čas sekundách s 3 desetinnými místy = `sw.ElapsedMilliseconds / 1000.0`

Měření času autor vložil těsně kolem výpočetní části kódu, aby nebyl čas ovlivněn dalšími faktory, jako je deklarace proměnných, zápis do souboru atd.

4.4.7 **Update UI**

4.4.7.1 Problémy v sériové implementaci

V sériové implementaci, kde celý kód běží v hlavním aplikačním threadu, však autor narazil na problém neresponzibility aplikace během výpočtu fraktálu.

Sériový návrh nemá za úkol řešit responzibilitu aplikace, ale aby bylo možné podat uživateli informace o běhu aplikace, bylo nutno vynutit překreslení prvku.

Existují tři možnosti, jak vynutit překreslení GUI prvku:

- Control.Invalidate(bool);
    - označí prvek pro překreslení, které proběhne, až se UI thread dostane k překreslování
- Control.Update();
    - vynutí překreslení prvku (přeskočí frontu zpráv)
    - v programu je pro využitý tento příkaz, překreslení proběhne jen při změně
- Control.Refresh();
    - zavolá Control.Invalidate(true) a Control.Update().

Příklad úpravy textu labelu:

```
labTestCount.Text = "Running test " + (testCountCompleted + 1)
       + " / " + testCountTotal;
labTestCount.Update(); //forces label to redraw
```

4.4.7.2 Komplikace v asynchronní (a paralelní) implementaci

Změny v UI v prostředí Microsoft C# jsou možné výhradně z hlavního aplikačního threadu (nazvěme ho pro naše potřeby UI threadem). Z ostatních vláken se přístup k UI řeší spuštěním delegátu přes příkaz Control.Invoke(Delegát).

Protože v případě sériové implementace je možné na UI prvky přistupovat přímo, mohl by příklad přístupu k prvku UI vypadat takto:

```
if (labRunStatus.InvokeRequired)
        labRunStatus.Invoke(new Action(() =>
        {
                labRunStatus.Text = "Starting calculations";
        }));
else
        labRunStatus.Text = "Starting calculations";
```

Testováním bylo zjištěno, že volání Control.Invoke i z UI threadu (kde není potřeba) nezpomaluje aplikaci, proto se autor kvůli čistšímu kódu rozhodl vždy použít invoke:

```
labRunStatus.Invoke(new Action(() =>
        labRunStatus.Text = "Starting calculations"));
```

4.4.8 **Dynamické labely pro uživatele**

Pro přehlednost zobrazeny bez Control.Invoke, v kódu jsou však vždy invokovány viz kapitola 4.4.7.2.

Aktuální množství spočítaných bodů v průběhu výpočtů:

```
calculaionPoints += Convert.ToInt64(
        Math.Floor(((_xMax - _xMin) / _stepSize) + 1));
labRunStatus.Text = "Running: "
        + calculaionPoints.ToString("000 000 000") + " points done";
labRunStatus.Update();
```

Výsledné hodnoty jednoho testu:

```
labRunStatus.Text = "Finished [" + sw.ElapsedMilliseconds / 1000.0 + " s]"
+ "\npoints total: " + (Math.Floor(((_xMax - _xMin) / _stepSize) + 1) *
Math.Floor(((_yMax - _yMin) / _stepSize) +1)).ToString("### ### ### ### ###")
+ "\ncalculations: " + calculationsTotal.ToString("### ### ### ### ###")
```

4.4.9 Zápis výsledků do souboru

Na konci každého testu, tj. na konci smyčky dávky testů (viz 4.4.2), chci zapsat naměřené hodnoty do souboru pro další srovnání a statistické zpracování.

Formát souboru autor zvolil hodnoty oddělené středníkem, inspirované formátem CSV. Jeho struktura souboru byla navržena následovně.

> *typ paralelnosti;úroveň souběžnosti;_iteractionsMax;_stepSize;čas v sekundách; pořadí v dávce;velikost dávky*

Při spuštění aplikace se kontroluje existence složky Results v rootu aplikace:

```
private void Form1_Load(object sender, EventArgs e)
{
	System.IO.Directory.CreateDirectory("Results");
```

Také jsou deklarovány globální privátní proměnné

```
//promenne pro zapis vysledku do souboru
private int testCountTotal = 1;
private int testCountCompleted = 0;
private string testModeName;
private int testTasksNum;
```

Při stisku tlačítka pro generování fraktálu se uloží string testModeName a úroveň souběžnosti – množství paralelních tasků vykonávajících výpočet fraktálu:

```
private void btnGenerate_Click(object sender, EventArgs e)
{
	//nastavit zakladni hodnoty pro zapis do souboru
	testCountTotal = Convert.ToInt32(maskedTxtBxTestCount.Text); //nemuze
	hodit exception, pole muze obsahovat jen int
	if (radioMainThread.Checked)
	{
		testModeName = "UI";//UI thread
		testTasksNum = 1;
	} else if (radioAsync.Checked)
	{
		testModeName = "Async";//second thread for responsibility
		testTasksNum = 1;
	} else if (radioParallelFor.Checked)
	{
		testModeName = "Par";//parallel for tasks
		testTasksNum = Convert.ToInt32(
			listBoxConcurrency.SelectedValue.ToString());
	}
```

Samotný zápis výsledků probíhá na konci každého úspěšného vygenerování Mandelbrotovy množiny.

Blok using zajistí, že se objekt po jeho použití korektně zlikviduje pomocí Dispose. (Hanák, 2009) ("using Statement (C# Reference)", 2015)

```
//zapis prave dokonceneho testu do souboru
using (System.IO.StreamWriter file =
       new System.IO.StreamWriter(@"Results\myResults_" + testModeName +
       testTasksNum + "_i" + _iterationsMax + "_s" + _stepSize + ".txt", true))
       //true=data pripisuji, neprepisuji
{
       file.WriteLine(testModeName + ";" + testTasksNum + ";"
       + _iterationsMax + ";" + _stepSize + ";"
       + (sw.ElapsedMilliseconds/1000.0) + ";"
       + (testCountCompleted+1).ToString("0000") + ";"
       + testCountTotal.ToString("0000"));
}
```

4.5 Implementace asynchronního responzivního návrhu

Využitím technologií popsaných v kapitole 3.2.2.5 byl vytvořen dvojitý wrapper nad sériovou metodou `MandelbrotMainThreadSync`, která je obsahem celé kapitoly 4.4.

```
#region PARALLEL ASYNC AWAIT Task.Run
private async void CallMandelbrotMainThreadAsync()
{
	await MandelbrotMainThreadTask();
}
private Task MandelbrotMainThreadTask()
{
	return Task.Run(
		() => MandelbrotMainThreadSync()
	);
}
#endregion
```

4.6 Implementace návrhu paralelizace pomocí Task Library

Využitím technologií popsaných v kapitole 3.2.2.5 byl vytvořen dvojitý wrapper nad metodou `MandelbrotParallelFor`, která je téměř identická jako sériové metoda `MandelbrotMainThreadSync` kromě změn popsaných v 4.4.4.3.

```
//------------------------------------------------------------------
#region PARALLEL FOR
private async void CallMandelbrotParallelForAsync()
{
	await MandelbrotParallelForTask();
}
private Task MandelbrotParallelForTask()
{
	return Task.Run(
		() => MandelbrotParallelFor()
	);
}
//------------------------------------------------------------------
private void MandelbrotParallelFor()
{
	(obsah viz popisek výše)
}
private static IEnumerable<double> Iterate(…){…}
#endregion
```

5 Výsledky a diskuse

5.1 Výsledky

5.1.1 Testování

Struktura ukládaných dat je autorem podrobně popsána v kapitole 4.4.9.

Zaznamenávaná data se skládají z parametrů vstupujících do algoritmu (podle kterých se budou výsledky „seskupovat“), z nastavení módu paralelity, dále kolik testů je v aktuálně spuštěné dávce testů a chronologické pořadí dokončeného testu, ale nejdůležitější naměřená informace je samozřejmě čas vygenerování kompletního fraktálu dle zadaných parametrů. Použitý diagnostický nástroj je v kapitole 4.4.6.

Informace o každém naměřeném testu se vždy přidá do souboru označeného módem paralelismu a vstupními parametry. To znamená, že opětovné spuštění testování stejným módem programu a se stejnými vstupními parametry, bude rozšiřovat předešlý soubor výsledků.

Testování bylo prováděno na počítači Lenovo Ideapad Y500 s procesorem Intel® Core™ i7-3630QM CPU @ 2.40Hz (4 jádra, 8 logických procesorů), 16GB paměti RAM, na 64-bitovém systému Windows 10. Autor měl snahu, aby při testování neprobíhala žádná jiná náročná aktivita, která by mohla ovlivnit naměřené hodnoty.

Tabulka 1 Testování zabralo 65,74 hodin čistého času

Row Labels	Sum of time	
10	**76519,917**	**21,26**
0,0015	76519,917	
200	**75106,736**	**20,86**
0,0025	75106,736	
2500	**85034,134**	**23,62**
0,0075	85034,134	
Grand Total	**236660,787**	**65,74**
	[sekundy]	[hodiny]

Algoritmus běžel dohromady téměř 66h času. Měření času však měří pouze matematický algoritmus každého bodu, reálný čistý celkový čas běhu testů tedy bude delší o jiné části kódu, například renderování. Také do času testování lze započítat ruční spouštění.

5.1.2 **Výsledky**

5.1.2.1 Vstupní parametry – zjednodušená interpretace

Maximální počet iterací *(i)* přidává výpočetní náročnost vnitřnímu algoritmu. Čím větší, tím pracnější výpočet každého bodu.

Číslo stepSize (s) udává hustotu bodů na ploše. Čím menší velikost kroku, tím více bodů se bude počítat.

Testy byly prováděny na těchto třech kombinacích vstupních parametrů:

i = 10, s = 0,0015 - malá výpočetní náročnost každého bodu; hodně bodů

i = 200, s = 0,0025 - střední výpočetní náročnost každého bodu; o něco méně bodů

i = 2500, s = 0,0075 - vysoká výpočetní náročnost každého bodu; málo bodů

5.1.2.2 Vysvětlení pojmenování

UI_1 – synchronní aplikace, kde výpočet běží v UI Threadu

Async_1 – asynchronní responsivní řešení

Par_# – paralelní implementace kde # značí množství vytvořených tasků

Par_-1 – označení paralelní implementaci s neomezenou úrovní souběžnosti

5.1.2.3 Nástroj pro analýzu

Data byla zpracována pomocí kontingenčních tabulek a grafů v programu Excel.

5.1.2.4 Množství dat

Tabulka 2 Soubor dat obsahuje 15633 měření

Count of seqT	Column Labels			
Row Labels	**10**	**200**	**2500**	**Grand Total**
Async_1	1000	970	800	2770
Par_1	1011	500	800	2311
Par_-1	500	647	500	1647
Par_16		500		500
Par_2	1000	500	800	2300
Par_4	1000	500	800	2300
Par_8	500	500	500	1500
UI_1	1000	500	805	2305
Grand Total	**6011**	**4617**	**5005**	**15633**

5.2 Diskuse

V kapitole 4.1.1 autor vyslovil předpoklad, u vhodně navržené aplikace pomůže paralelizace jejího algoritmu ke zrychlení běhu. Tento předpoklad zkusíme ověřit praktickým testem provedeném na dostatečně velkém množství dat (viz předchozí kapitola).

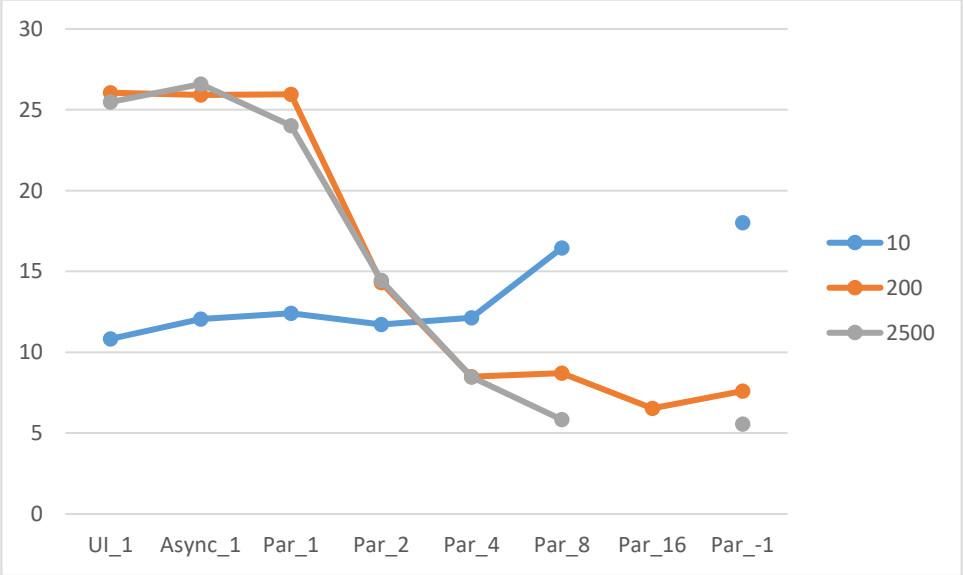
Tabulka 3 Naměřené průměry časů dle Iterací a dle použitého algoritmu

Průměr časů	iterace (výpočetní náročnost)		
Algoritmus	10	200	2500
UI_1	10,827365	26,05769	25,47670807
Async_1	12,054227	25,91709485	26,5871425
Par_1	12,41492977	25,961504	24,0233325
Par_2	11,713681	14,315998	14,4596525
Par_4	12,133915	8,497246	8,4664025
Par_8	16,461248	8,716154	5,831278
Par_16		6,534874	
Par_-1	18,017222	7,612706337	5,561042

Testování souběžnosti 16 na 8 threadovém počítači nebylo efektivní.

Rozborem tabulky zjistíme, že značné zrychlení (téměř 500%) je znát na výpočetně složitých operacích, zatímco použitím přílišného souběžného zpracování informací na výpočetně nenáročné operace může způsobit, že prostředky pro správu paralelismu budou větší, než přínos z paralelního výpočtu. Takový výsledek je jasně vidět v prvním sloupci dat, kdy rozeběhnutí algoritmu s defaultní úrovní souběžnosti (maximální) oproti synchrnonnímu řešení snížilo rychlost na 60%.

Tabulka 4 Graf porovnání výhodnosti vícevláknových komputací v závislosti na nich



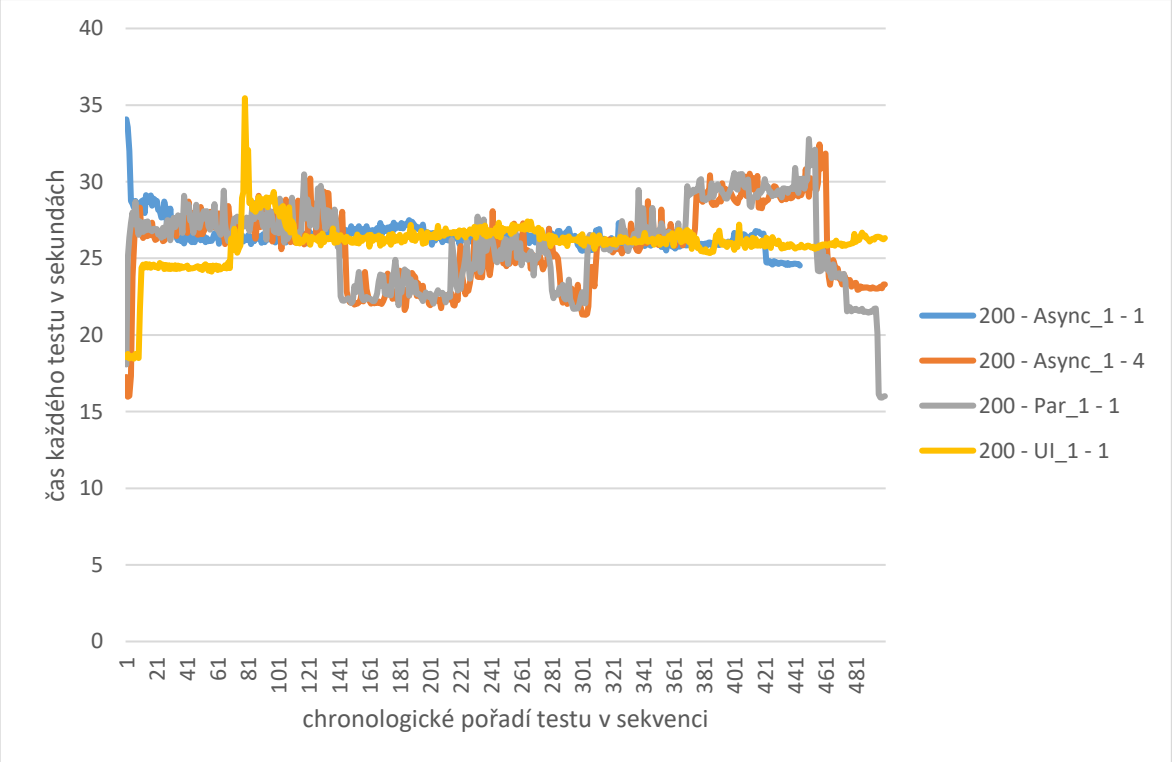
Jak je vidět i na grafu (viz výše), dostatečně výpočetně náročné operace (i=200 a i=2500) se s větší mírou paralelizace zrychlují. Pro řešení výpočetně nenáročné aplikace však přílišná výpočetní souběžnost není vhodná, čímž se potvrzuje autorův předpoklad.

Dále autor chtěl ověřit podezdření, zda při tak velikém množství identických matematických operací neprovádí systém nějakou optimalizaci či cachování, které by ovlivňovalo navazující testy.

K ověřování byly porovnávány chronologicky seřazené křivky mnoha testovaných dávek za účelem najít rostoucí či klesající tendenci. Na začátku těchto testování trvá několik jednotek testů, než se ustálí rychlost testování, tato změna však probíhá skokově a neindikuje žádnou tendenci.

Až na lokální výkyvy pravděpodobně způsobené nestále vytíženým prostředím tato data tvoří konstantní přímky, čímž byla myšlenka vyvrácena. Následující graf ukazuje jednu z proběhlých kontrol.

Tabulka 5 Hledání tendence v průběhu dávek testů



5.3 Zhodnocení

Prapůvodní myšlenkou účelu práce bylo nastudovat téma s praktickým využitím, které autora zajímalo a ještě ho z důvodu jeho rozsáhlosti a komplexnosti neovládá. Toto téma by se dalo nazvat využití paralelismu v praxi. Hlavním cílem tedy bylo mít silnou praktickou část, efektivní a elegantní paralelní algoritmus, jehož vývoj dodá autorovi znalosti a zkušenosti k tomu, používat paralelismus i v běžné praxi.

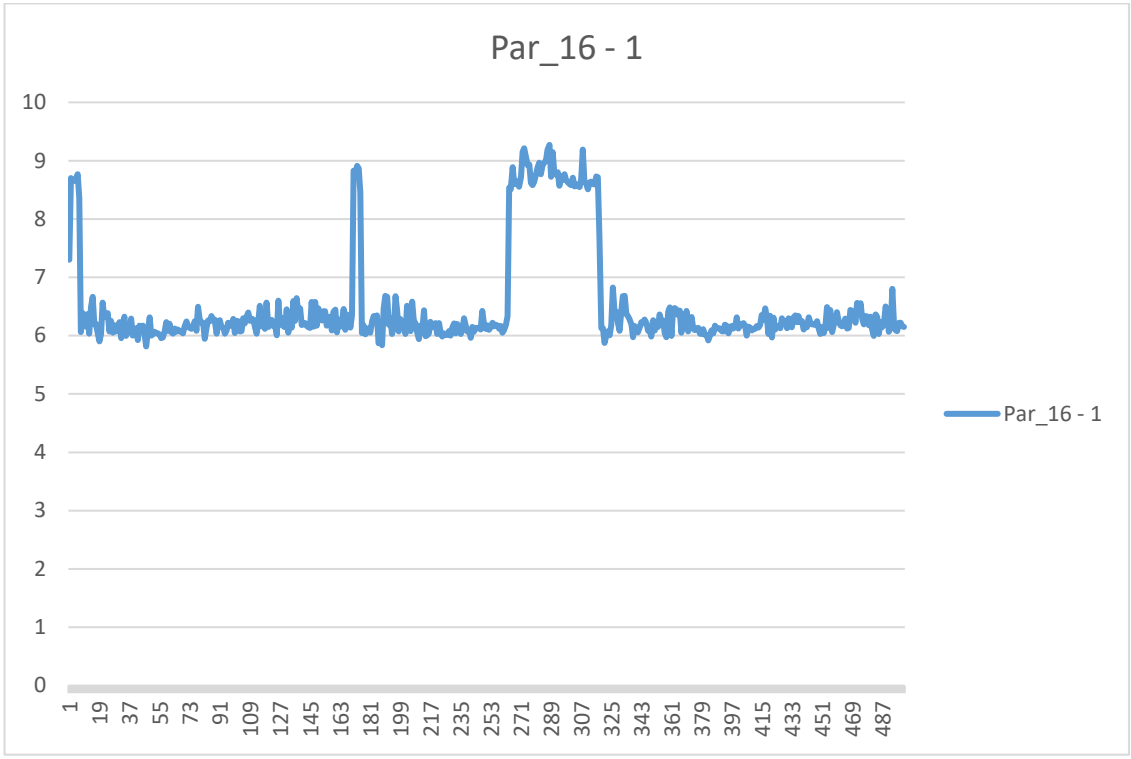
Z tohoto hlediska autor hodnotí práci velmi optimisticky. Podařilo se nastudovat množství odborné literatury zabývající se tímto problémem a sepsat o tom rozsáhlou literární rešerši. S úspěchem se setkalo splnit cíl, naučit se správně využívat paralelismus .NET frameworku a s těmito znalostmi naimplementovat program podle cílových představ.

Výsledná aplikace je dostatečně parametrická na to, aby se na ní dalo testovat a poskytla širokou škálu dat pro porovnání paralelních a sériových přístupů. Data se podařilo vygenerovat hojné množství (kap. 5.1.2.4), interpretovat (kap. 5.2) je a konstatovat, že data prokázali původní předpoklad (kapitola 4.1.1.).

Zde by autor rád řekl, že ho vypracovávání této práce skutečně bavilo, ať už prozkoumávání krás matematických vzorců a obrazců na cestě k pochopení fraktálů či řešení algoritmických oříšků ve snaze udržet kód elegantní. Pravděpodobně to autora podnítí k dalšímu samostudiu těchto problematik.

Poznámka k testování - Je vhodné si uvědomit, že testování výkonu aplikace na osobním počítači není úplně ideální. I když byla vynaložena všechna snaha, aby počítač během testování neprováděl jiné operace, nelze vyloučit rušení testování v důsledku programů na pozadí. Důkaz tohoto chování se dá najít přímo v jich probíraných datech. Následující graf představuje dávku 500 testů generování Mandelbrot setu, na y souřadnici znázorněn čas. Z grafu je jasně vidět, že algoritmus, který trvá programu konstantních 6 sekund se na 60 testů (9s*60=9minut) o 50% skokově zpomalil a následně vrátil na konstantní rychlost.

Tabulka 6 Ovlivňování testů programy na pozadí



Stálo by za to zmíňit jednu slabinu, a to, že i když je v aplikaci algoritmus implementován paralelně, přistupuje se uvnitř na sdílený prostředek kterým je buffer. Paralelní přístup na sdílený prostředek je vždy potřeba ošetřit. V programu je situace vyřešena jednoduchým příkazem lock (viz kapitola 3.1.4). Teoreticky by mohly být v průběhu algoritmu některá vlákna blokovaná čekajíc na uvolnění zámku a tím by se aplikace zpomalovala, v reálu je přístup do bufferu velmi krátký a sledování využití prostředků v průběhu testování naznačují, že se vlákna vzájemně nezdržují.

Stojí za zamyšlení zkusit si v budoucnu implementovat algoritmus bez použití sdílených prostředků a porovnat je. Je však možné, že by se přišlo na to, že takto získané prostředky ztratí zase v jiné části algoritmu.

6 Závěr

Cílem této práce byl primárně vývoj vícevláknové paralelní aplikace a implementovat algoritmus, který bude schopen běžet v sériové i paralelní verzi. Aplikace se podařila naimplementovat právě tak, jak autor zamýšlel.

Byly využity nejnovější technologie .NET Framework (viz kapitola 3.2.2.4 a 3.2.2.5). Sériové jádro programu (viz kap. 4.4.4) je naimplementované velmi podobně svému paralelnímu protějšku (popsané v kap. 4.4.4.3 a 4.6).

Program je dále schopen generovat i vizualizovat Mandelbrotovu množinu a zapisovat statistiky rychlosti generace v různých režimech paralelizace do souboru.

Výsledky např. ukázaly, že i dlouhá dávka testů má stále lineární cenu času. Hlavním výstupem výsledků je potvrzení autorova předpokladu, že paralelní programování má význam jen od určitého stupně výpočetní náročnosti jednotlivých vláken. Pokud se práce rozdělí až moc souběžně, náklady na správu paralelismu převáží získaný výpočetní výkon.

7 Seznam použitých zdrojů

"Asynchronous Programming with async and await C#". 2015. *Microsoft Developer Network.* [Online] 20. 7 2015. [Citace: 3. 3 2017.] https://msdn.microsoft.com/en-us/library/mt674882.aspx.

"How to: Write a Simple Parallel.ForEach Loop". 2012. *Microosft Developer Network.* [Online] 2012. [Citace: 27. 2 2017.] https://msdn.microsoft.com/en-us/library/dd460720(v=vs.110).aspx.

"Intro to the C# Language and the .NET Framework". 2015. Introduction to the C# Language and the .NET Framework. *Microsoft Developer Network.* [Online] 2015. [Citace: 11. 12 2016.] https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx.

"Lambda Expressions (C# Programming Guide)". 2015. *Microsoft Developer Network.* [Online] 20. 7 2015. [Citace: 21. 2 2017.] https://msdn.microsoft.com/en-us/library/bb397687.aspx.

"Lambda Expressions in PLINQ and TPL". 2010. *Microsoft Developer Network.* [Online] 2010. [Citace: 22. 2 2017.]

"lock Statement (C# Reference)". 2015. *Microsoft Developer Network.* [Online] 20. 7 2015. [Citace: 4. 3 2017.] https://msdn.microsoft.com/en-us/library/c5kehkcz.aspx.

"Task-based Asynchronous Programming". 2012. *Microsoft Developer Network.* [Online] 2012. [Citace: 24. 2 2017.] https://msdn.microsoft.com/en-us/library/dd537609.aspx.

"using Statement (C# Reference)". 2015. *Microsoft Developer Network.* [Online] 20. 7 2015. [Citace: 11. 3 2017.] https://msdn.microsoft.com/en-us/library/yh598w02.aspx.

"What's New in the .NET Framework 4". 2012. *Microsoft Developer Network.* [Online] 3 2012. [Citace: 19. 1 2017.] https://msdn.microsoft.com/en-us/library/ms171868(v=vs.100).aspx.

Avidar, Dan. 2013. Understand Lambda Expressions. *Code Project.* [Online] 5. 3 2013. [Citace: 21. 2 2017.] https://www.codeproject.com/tips/298963/understand-lambda-expressions-in-minutes.

Blaise, Barney. 2016. Introduction to Parallel Computing. *Livermore computing center.* [Online] Lawrence Livermore National Laboratory, 2016. [Citace: 16. 1 2017.] https://computing.llnl.gov/tutorials/parallel_comp/. UCRL-MI-133316.

Bray, Brandon. 2012. Async in 4.5: Worth the Await. *Microsoft Developer Network Blogs.* [Online] 3. 4 2012. [Citace: 3. 3 2017.] https://blogs.msdn.microsoft.com/dotnet/2012/04/03/async-in-4-5-worth-the-await/.

Hanák, Jan. 2009. *Praktické objektové programování v jazyce C# 4.0.* Brno : Artax, 2009. ISBN 978-80-87017-07-4.

Nagel, Christian, a další. 2009. *C# 2008.* Brno : Computer Press, 2009. ISBN 978-80-251-2401-7.

Pirzada, Usman. 2015. Intel ISSCC: 14nm all figured out, 10nm is on track, Moores Law still alive and kicking. *WCCFTECH.* [Online] 24. 2 2015. [Citace: 25. 11 2016.] http://wccftech.com/intel-isscc-14nm/.

Ranjan, Sen. 2008. Developing Parallel Programs. *Microsoft Development Network.* [Online] 2008. [Citace: 16. 1 2017.] https://msdn.microsoft.com/en-us/library/cc983823.aspx#Parallel.

Sharp, John. 2010. *Microsoft Visual C# 2010: krok za krokem.* Brno : Computer Press, 2010. ISBN 978-80-251-3147-3.

Stellman, Andrew a Greene, Jennifer. 2013. *Head first C#.* Sebastopol : O'Reilly Media, 2013. ISBN 978-1-449-34350-7.

Toub, Stephen. 2009. Parallel For Loops over Non-Integral Types. *Microsoft Developer Network Blogs.* [Online] 24. 6 2009. [Citace: 8. 3 2017.]

—. 2011. Parallel Programming with .NET. *Microsoft Developer Network Blogs.* [Online] 24. 10 2011. [Citace: 7. 3 2017.] https://blogs.msdn.microsoft.com/pfxteam/2011/10/24/task-run-vs-task-factory-startnew/.
Weisstein, Eric W. 2009. Mandelbrot Set. *MathWorld--A Wolfram Web Resource.* [Online] 6. 2 2009. [Citace: 18. 10 2016.] http://mathworld.wolfram.com/MandelbrotSet.html.