Microsoft

Ján Hanák

Praktické paralelné programovanie v jazykoch C# 4.0 a C++

intel

Microsoft® Visual Studio® 2008

Microsoft® Visual Studio®

Ján Hanák

Praktické paralelné programovanie v jazykoch C# 4.0 a C++

Artax
2009

Autor: Ing. Ján Hanák, MVP

Praktické paralelné programovanie v jazykoch C# 4.0 a C++

Recenzenti:	doc. RNDr. Jozef Fecenko, CSc. Ing. Magdaléna Cárachová, PhD.
Vydanie:	prvé
Rok prvého vydania:	2009
Náklad:	500 ks
Jazyková korektúra:	Ing. Peter Kubica
Vydal:	Artax a.s., Žabovřeská 16, 616 00 Brno pre Microsoft s.r.o., Vyskočilova 1461/2a, 140 00 Praha 4
Tlač:	Artax a.s., Žabovřeská 16, 616 00 Brno
ISBN:	978-80-87017-06-7

Obsah

Úvod 3

Poďakovanie 7

Venovanie 7

1 Charakteristika praktických ukážok paralelného programovania 8

1.1 Konfigurácie počítačových systémov určených na empirické testovanie sekvenčných a paralelných programov 12

2 Praktická ukážka č. 1: Matematické operácie s 3D vektormi 13

2.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti 21

3 Praktická ukážka č. 2: Riešenie masívnych súprav sústav 3 lineárnych rovníc s 3 neznámymi 25

3.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti 32

4 Praktická ukážka č. 3: Lineárna algebra – Násobenie matíc typu 300x300 36

4.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti 44

5 Praktická ukážka č. 4: Paralelné grafické transformácie bitových máp (paralelizmus s vysokou úrovňou abstrakcie) 48

5.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti 54

6 Praktická ukážka č. 5: Paralelné grafické transformácie bitových máp (paralelizmus s nízkou úrovňou abstrakcie) 58

6.1 Empirické testovanie paralelného programu a kvantifikácia nárastu výkonnosti 63

6.2 Diagnostika a monitorovanie výkonu paralelného programu profilovacím programom integrovaným v prostredí produktu Visual Studio 2010 68

7 Praktická ukážka č. 6: Vyhľadávanie prvočísel (riadený a natívny paralelizmus).. 75

7.1 Empirické testovanie riadeného sekvenčného programu, natívneho sekvenčného programu, riadeného paralelného programu a natívneho paralelného programu s kvantifikáciou nárastu výkonnosti 82

8 Praktická ukážka č. 7: Numerická integrácia - natívny paralelizmus pomocou rozhrania OpenMP 91

8.1 Empirické testovanie sekvenčného a paralelného natívneho programu a kvantifikácia nárastu výkonnosti 94

8.2 Diagnostika a monitorovanie výkonu paralelného natívneho programu profilovacím programom Intel Parallel Amplifier integrovaným v prostredí produktu Intel Parallel Studio 98

9 Praktická ukážka č. 8: Implicitný natívny paralelizmus - automatická paralelizácia algoritmu numerickej integrácie pomocou prekladača Intel C++ Compiler 11.1 102

10 Finálny sumár kvantitatívnych charakteristík praktických ukážok 109

11 Metodika riešiaca vývoj paralelných programov optimalizovaných pre beh na počítačových systémoch s viacjadrovými procesormi 111

11.1 Metodika riešiaca paralelizáciu pôvodne sekvenčného programu 112

11.2 Metodika riešiaca vývoj úplne nového paralelného programu 124

Záver 128

O autorovi 129

Úvod

Vážené čitateľky, vážení čitatelia,

po mimoriadne pozitívnej akceptácii vysokoškolskej učebnice „Základy paralelného programovania v jazyku C# 3.0“[1] sme usúdili, že tematika paralelného a paralelného objektovo orientovaného programovania (POOP) je poslucháčmi informatických vysokých škôl univerzitného typu, akademickými pracovníkmi, komerčnými vývojármi a fanúšikmi programovania ponímaná ako veľmi atraktívna a žiadaná. Keďže chceme v osvete paradigmy POOP prostredníctvom najmodernejších softvérových technológií pokračovať i naďalej, prichádzame s novou knihou, ktorá sa sústreďuje na praktické paralelné programovanie. Cieľom predkladanej publikácie je predstaviť základy POOP na praktických ukážkach, ktoré boli vyvinuté v programovacích jazykoch C# 4.0 a C++ (podľa štandardu C++0x), s využitím vývojových prostredí Microsoft Visual Studio 2008, Microsoft Visual Studio 2010 a Intel Parallel Studio.

Vzhľadom na to, že kniha „Praktické paralelné programovanie v jazykoch C# 4.0 a C++“ priamo nadväzuje na dielo „Základy paralelného programovania v jazyku C# 3.0“, predpokladáme, že čitatelia ovládajú elementárne princípy, postuláty a techniky paralelného programovania v rozsahu, v akom sme ich vysvetlili v predchádzajúcej vysokoškolskej učebnici. K ďalším prerekvizitám, ktoré u čitateľov očakávame, patrí solídne zvládnutie programovacích jazykov C# a C++ na pokročilej úrovni.

Na najvyššej úrovni abstrakcie môžeme všetky praktické ukážky tejto knihy rozdeliť do 2 samostatných skupín:

1. **Praktické ukážky demonštrujúce implementáciu explicitného riadeného paralelizmu.** Tieto praktické ukážky sme vyvinuli

[1] Hanák, J.: Základy paralelného programovania v jazyku C# 3.0. Brno: Artax, 2009. Vysokoškolská učebnica je v elektronickej podobe zdarma k dispozícii na nasledujúcej adrese: http://msdn.microsoft.com/cs-cz/dd727769.aspx.

v programovacom jazyku C# 4.0 s podporou knižnice Task Parallel Library (TPL), ktorá je súčasťou bázovej knižnice tried (BCL) vývojovo-exekučnej platformy Microsoft .NET Framework 4.0. Ako hlavné vývojové prostredie sme použili Microsoft Visual Studio 2010.

2. **Praktické ukážky demonštrujúce implementáciu explicitného natívneho paralelizmu.** Tieto praktické ukážky sme vytvorili v programovacom jazyku C++ s podporou nových syntakticko-sémantických inovácií, ktoré zavádza pripravovaný ISO štandard so zatiaľ pracovným označením C++0x. Okrem najaktuálnejšej verzie jazyka C++ s výhodou uplatňujeme programové konštrukcie a entity z knižníc Microsoft Parallel Patterns Library (PPL) a OpenMP. Citeľný nárast pracovnej produktivity pri tvorbe natívnych paralelných programov sme zaznamenali pri použití kombinácie produktov Microsoft Visual Studio 2008 a Intel Parallel Studio. Produkt Intel Parallel Studio je balíkom nástrojov, ktoré umožňujú efektívne absolvovať všetky náročné štádiá návrhu, tvorby a ladenia paralelných natívnych aplikácií napísaných v jazykoch C a C++.

Poznámka: Keďže v čase tvorby tohto diela sa produkt Intel Parallel Studio integroval do prostredia produktu Microsoft Visual Studio 2008, použili sme v tých častiach knihy, ktoré sa produktu Intel Parallel Studio venujú, spomínané staršie vývojové prostredie spoločnosti Microsoft. (Ako však býva u spoločnosti Intel dobrým zvykom, verzia nástroja Parallel Studio pre Visual Studio 2010 sa objaví zakrátko po uvedení finálnej verzie vývojového prostredia od firmy Microsoft.)

V knihe sa nachádzajú nasledujúce praktické ukážky:

1. **Matematické operácie s 3D vektormi.**
   Technológie: C# 4.0 a TPL.

2. **Riešenie masívnych súprav sústav 3 lineárnych rovníc s 3 neznámymi.**
   Technológie: C# 4.0 a TPL.

3. **Lineárna algebra – Násobenie matíc typu 300x300.**
   Technológie: C# 4.0 a TPL.

4. **Paralelné grafické transformácie bitových máp (paralelizmus s vysokou úrovňou abstrakcie).**
   Technológie: C# 4.0 a TPL.

5. **Paralelné grafické transformácie bitových máp (paralelizmus s nízkou úrovňou abstrakcie).**
   Technológie: C# 4.0 a BCL.

6. **Vyhľadávanie prvočísel (riadený a natívny paralelizmus).**
   Technológie: C# 4.0 a TPL, C++0x a PPL.

7. **Numerická integrácia – explicitný natívny paralelizmus.**
   Technológie: C++0x a OpenMP.

8. **Numerická integrácia – implicitný natívny paralelizmus.**
   Technológie: C++0x, automatický paralelizér prekladača Intel C++ 11.1.

Každá praktická ukážka je vybavená informačným panelom, ktorý podáva nasledujúce informácie:

- Cieľ praktickej ukážky.
- Vedomostná náročnosť praktickej ukážky.[2]
- Časová náročnosť praktickej ukážky[3].
- Softvérové technológie, ktoré boli použité pri vytváraní praktickej ukážky.
- Druh paralelizmu, ktorý praktická ukážka implementuje.

[2] Úroveň vedomostnej náročnosti je determinovaná relatívne s ohľadom na cieľové publikum knihy.
[3] Úroveň časovej náročnosti je determinovaná relatívne s ohľadom na cieľové publikum knihy.

Pri tvorbe praktických ukážok sme uplatnili metodiku, ktorú detailne predstavujeme v kapitole *1 Charakteristika praktických ukážok paralelného programovania*. Sekvenčné a paralelné programy, ktoré v jednotlivých praktických ukážkach konštruujeme, sme podrobili výkonnostným testom na variabilne dimenzovaných počítačových systémoch (ich bližší opis uvádzame v kapitole *1.1 Konfigurácie počítačových systémov určených na empirické testovanie sekvenčných a paralelných programov*). Našou intenciou bolo vždy kvantifikovať nárast výkonnosti paralelných programov, ktoré vznikli paralelizáciou pôvodne sekvenčných programov.

Najväčšou konkurenčnou výhodou tejto knihy je nová metodika, ktorá rieši vývoj paralelných programov optimalizovaných pre beh na počítačových systémoch s viacjadrovými procesormi. S využitím tejto metodiky môžu softvéroví vývojári maximalizovať svoju produktivitu pri paralelizácii existujúcich sekvenčných programov, či pri vývoji úplne nových paralelných programov.

Ján Hanák

Bratislava, november 2009

Poďakovanie

Ako autor tejto knihy by som chcel vyjadriť svoje poďakovanie recenzentom, doc. RNDr. Jozefovi Fecenkovi, CSc., a Ing. Magdaléne Cárachovej, PhD., za dôkladné posúdenie tohto diela a hodnotné námety na jeho ďalšie skvalitnenie.

Veľká vďaka patrí rovnako Mgr. Miroslavovi Kubovčíkovi zo spoločnosti Microsoft Slovakia za výbornú podporu, ústretový prístup a skvelú niekoľkoročnú spoluprácu, výsledkom ktorej je aj táto kniha (a ako obaja veríme, aj mnoho ďalších diel, ktoré pre vývojárov pripravíme v nasledujúcich rokoch).

Moje poďakovanie a úcta patrí aj Ing. Petrovi Ulvrovi zo spoločnosti Intel za výborné softvérové zabezpečenie našej spolupráce.

V neposlednom rade by som sa rád poďakoval aj Ing. Martinovi Lukáškovi a Ing. Vítovi Obůrkovi zo spoločnosti Artax za veľkolepú a vždy hladkú spoluprácu pri príprave grafických materiálov, zlome a tlači tejto knihy.

Venovanie

Túto knihu venujem všetkým nadaným ľuďom,
ktorí každodenne vynakladajú svoju energiu, úsilie a čas na to,
aby sa z tohto sveta stalo lepšie miesto pre život.

1 Charakteristika praktických ukážok paralelného programovania

Táto kniha je venovaná praktickým ukážkam paralelného programovania. Predstavené praktické ukážky majú nasledujúce charakteristiky:

1. Riešime parciálne problémy, respektíve inštancie parciálnych problémov.

2. Konštruujeme sekvenčné a paralelné algoritmy, ktoré riešia parciálne problémy. Pritom sa sústreďujeme na analýzu nárastu výkonnosti pôvodne sekvenčných programov po ich paralelizácii.

   Celý proces prebieha v týchto krokoch:

   - Najskôr zostavíme sekvenčný algoritmus, ktorý rieši parciálny problém.

   - Sekvenčný algoritmus implementujeme do sekvenčného programu.

   - Empirickými testami zisťujeme výkonnostné metriky sekvenčného programu. Empirické testy sekvenčného programu sú realizované na počítačoch s viacjadrovými procesormi[4]. Testy skúmajú dĺžku exekučného času, ktorý je nutný na spracovanie sekvenčného programu. Každý test sa skladá z piatich relácií, v rámci ktorých zaznamenáme dĺžku exekučných časov spracovania sekvenčného programu. Potom vypočítame priemerný exekučný čas spracovania sekvenčného programu.

[4] Presné technické konfigurácie testovacích počítačových systémov sú uvedené v časti *1.1 Konfigurácie počítačových systémov určených na empirické testovanie sekvenčných a paralelných programov*.

- Vytvorený sekvenčný program podrobíme transformácii, ktorej výsledkom je ekvivalentný paralelný program. Skonštruovaný paralelný program vznikne paralelizáciou pôvodne sekvenčného algoritmu, ktorý implementuje sekvenčný program.

- Empirickými testami skúmame výkonnostné metriky paralelného programu. Empirické testy paralelného programu sú realizované na rovnakých počítačoch, na ktorých boli uskutočnené empirické testy sekvenčného programu. Podobne ako pri testoch sekvenčného programu, tak aj pri testovaní paralelného programu analyzujeme priemerný exekučný čas jeho spracovania (ktorý plynie z 5 relácií).

- Porovnávame výkonnostné charakteristiky sekvenčného a paralelného programu a determinujeme nárast výkonnosti paralelného programu vo vzťahu k sekvenčnému programu. Pritom používame nasledujúci matematický vzťah:

$$N_V = \frac{T_S}{T_P}$$

kde:

- N_V je nárast výkonnosti paralelného programu voči sekvenčnému programu.
- T_S je priemerný exekučný čas spracovania sekvenčného programu.
- T_P je priemerný exekučný čas spracovania paralelného programu.

Po určení exaktnej hodnoty nárastu výkonnosti paralelného programu prijímame záver o trende jeho výkonnosti, ktorý môže byť:

- Sublineárny, ak $N_V < n$.
- Lineárny, ak $N_V = n$.
- Superlineárny, ak $N_V > n$.

kde:

- n je počet exekučných jadier viacjadrového procesora.

- Vypočítavame efektivitu využitia výpočtových zdrojov počítačového systému podľa tohto matematického vzťahu:

$$e = \frac{N_V}{n} \times 100[\%]$$

kde:

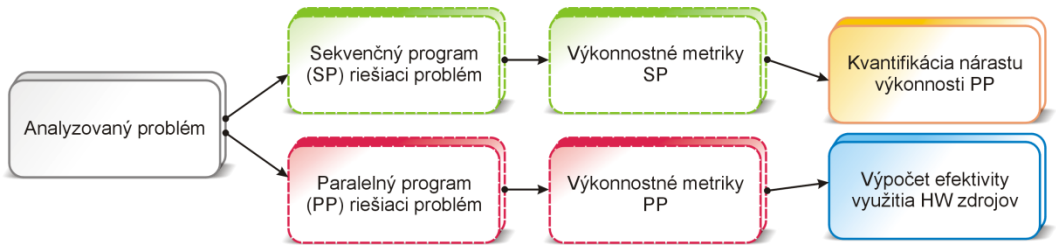
- e je efektivita využitia výpočtových zdrojov počítačového systému.
- N_V je nárast výkonnosti paralelného programu voči sekvenčnému programu.
- n je počet exekučných jadier viacjadrového procesora.

Efektivita je vyjadrená percentuálnou hodnotou, ktorá determinuje priemernú mieru využívania exekučných jadier viacjadrového procesora počas trvania životného cyklu paralelného programu. Ak napríklad zaznamenáme pri testovaní paralelného programu na počítači s 2-jadrovým procesorom nárast výkonnosti 1,8, efektivita využitia výpočtovej kapacity systému bude takáto:

$$e = \frac{1{,}8}{2} \times 100 = 90\ \%$$

Tento výsledok interpretujeme nasledujúcim spôsobom: V priemere využíval paralelný program v čase svojej exekúcie 90 % disponibilných výpočtových zdrojov počítačového systému. Každé exekučné jadro 2-jadrového procesora bolo v priemere 90 % času exekúcie paralelného programu zaneprázdnené vykonávaním strojového kódu tohto programu. Alebo opačne, v priemere bolo každé jadro 2-jadrového procesora 10 % času spracovania paralelného programu nečinné (zostávalo v neaktívnom stave). Prirodzene, optimálnym stavom je 100 % efektivita. Čím bližšie sa k tejto hodnote priblížime, tým je vytvorený paralelný program efektívnejší,

pretože maximalizuje mieru využívania hardvérovej platformy počítačového systému. V tejto súvislosti musíme uviesť, že koeficient efektivity môže nadobúdať aj vyššiu ako 100% hodnotu. S týmto javom sa stretávame pri paralelných programoch, ktoré preukazujú superlineárny nárast výkonnosti.



Obr. 1: Metodika použitá pri realizácii praktických ukážok paralelného programovania

3. Pri stavbe paralelných programov uplatňujeme spravidla koncepciu explicitného paralelizmu s rôznou úrovňou abstrakcie. Okrem jednej praktickej ukážky rozoberáme vždy explicitný paralelizmus s rôznou úrovňou abstrakcie (explicitný paralelizmus s nízkou, strednou a vysokou úrovňou abstrakcie).

4. Pri vytváraní sekvenčných a paralelných programov používame programovacie jazyky C# 4.0 a C++ (podľa štandardu ISO/IEC 14882:2003 a C++0x). V prípade jazyka C#4.0 pracujeme s jeho implementáciou v produkte Visual C# 2010 od spoločnosti Microsoft. C# 4.0 spolupracuje s bázovou knižnicou tried vývojovo-exekučnej platformy Microsoft .NET Framework 4.0. V prípade jazyka C++ používame implementáciu v produkte Visual C++ 2010. Jazyk C++ kooperuje s knižnicou jazyka C++, rovnako ako aj so štandardnou šablónovou knižnicou jazyka C++ (STL).

5. Všetky sekvenčné a paralelné programy sú štandardné konzolové aplikácie. Ich zdrojové kódy sú preložené v ostrých zostavovacích režimoch (Release). Vygenerované priamo spustiteľné súbory sú potom priamo spúšťané

z operačného systému a nie z vývojových prostredí príslušných programovacích jazykov.

1.1 Konfigurácie počítačových systémov určených na empirické testovanie sekvenčných a paralelných programov

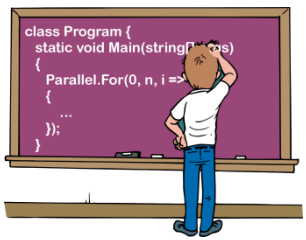
Za účelom empirického testovania výkonnosti vyvinutých sekvenčných a paralelných programov napísaných v programovacích jazykoch C# 4.0 a C++ používame tieto počítačové systémy:

1. Stolový počítač osadený 4-jadrovým procesorom Intel Core 2 Quad Q6600. Každé exekučné jadro viacjadrového procesora je taktované na 2,4 GHz a disponuje 2 MB vyrovnávacej pamäte druhej úrovne (L2-cache). Počítač obsahuje 4 GB vstavanej operačnej pamäte a beží na ňom 32-bitový operačný systém Microsoft Windows Vista.

2. Notebook vybavený 2-jadrovým procesorom AMD Turion 64 X2. Každé exekučné jadro viacjadrového procesora je taktované na 2,0 GHz a disponuje 512 KB L2-cache. Počítač spolupracuje s 2 GB operačnej pamäte a je na ňom nainštalovaný 32-bitový operačný systém Microsoft Windows 7.

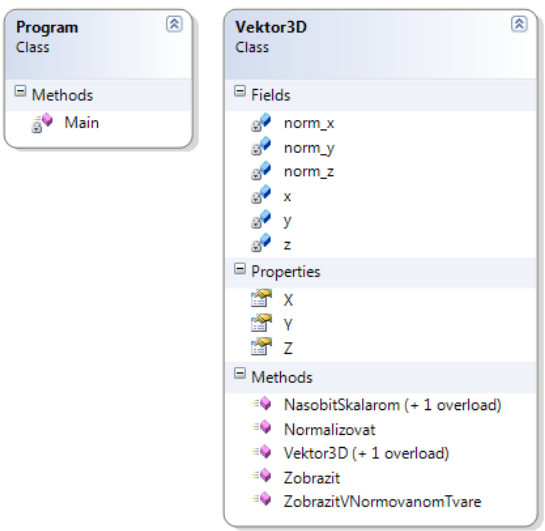
3. Notebook s nainštalovaným 2-jadrovým procesorom Intel Core 2 Duo T5800. Každé exekučné jadro viacjadrového procesora je taktované na 2,0 GHz a obsahuje 1 MB L2-cache. Počítač je vybavený 3 GB operačnej pamäte a 32-bitovým operačným systémom Microsoft Windows Vista.

2 Praktická ukážka č. 1: Matematické operácie s 3D vektormi



Cieľ praktickej ukážky:
Paralelizácia spracovania vybraných matematických operácií s trojrozmernými (3D) vektormi.
Vedomostná náročnosť: .
Časová náročnosť: 25 minút.
Softvérové technológie: C# 4.0 a TPL.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie: .

Vizualizácia tried sekvenčného programu:



Obr. 2: Diagram tried sekvenčného programu

Zdrojový kód sekvenčného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
```

```
using System.Text;
namespace diz
{
    // Deklarácia triedy, ktorá charakterizuje trojrozmerný (3D) vektor.
    class Vektor3D
    {
        // Definície dátových členov triedy.
        private double x, y, z;
        private double norm_x, norm_y, norm_z;
        // Definícia bezparametrického inštančného konštruktora.
        public Vektor3D()
        {
            x = y = z = 1.0;
        }
        // Definícia parametrického inštančného konštruktora.
        public Vektor3D(double x, double y, double z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        // Definícia bezparametrickej metódy na vynásobenie 3D vektora skalárom.
        public void NasobitSkalarom()
        {
            double skalar = 3.14 * Math.Asin(2 / 6.5) / 5.88 *
                (Math.Atan(1.55 / 1001));
            x *= skalar;
            y *= skalar;
            z *= skalar;
        }
        // Definícia parametrickej metódy na vynásobenie 3D vektora skalárom.
        public void NasobitSkalarom(double skalar)
        {
            x *= skalar;
            y *= skalar;
            z *= skalar;
        }
        // Definícia bezparametrickej metódy na normalizáciu 3D vektora.
        public void Normalizovat()
        {
            double velkostVektora = Math.Sqrt(x * x + y * y + z * z);
            norm_x = x / velkostVektora;
            norm_y = y / velkostVektora;
            norm_z = z / velkostVektora;
        }
        // Definícia bezparametrickej metódy na zobrazenie zložiek 3D vektora.
        public void Zobrazit()
```

```
        {
            Console.Write("\n[{0}, {1}, {2}]\n", x, y, z);
        }
        // Definícia bezparametrickej metódy na zobrazenie zložiek 3D vektora
        // v normalizovanom tvare.
        public void ZobrazitVNormovanomTvare()
        {
            Console.Write("\n[{0}, {1}, {2}]\n", norm_x, norm_y, norm_z);
        }
        // Definícia skalárnej inštančnej vlastnosti
        // pre x-ovú zložku 3D vektora.
        public double X
        {
            get { return x; }
            set { x = value; }
        }
        // Definícia skalárnej inštančnej vlastnosti
        // pre y-ovú zložku 3D vektora.
        public double Y
        {
            get { return y; }
            set { y = value; }
        }
        // Definícia skalárnej inštančnej vlastnosti
        // pre z-ovú zložku 3D vektora.
        public double Z
        {
            get { return z; }
            set { z = value; }
        }
    }
    class Program
    {

        static void Main(string[] args)
        {
            // Špecifikácia počtu 3D vektorov, s ktorými budeme uskutočňovať
            // matematické operácie.
            const uint pocetVektorov = 10000000;
            // Vytvorenie poľa 3D vektorov.
            Vektor3D[] vektory = new Vektor3D[pocetVektorov];
            Random generator = new Random();
            Stopwatch stopky = new Stopwatch();
            stopky.Start();
            Console.WriteLine("Prebieha vytváranie vektorov...");
            // Inicializácia 3D vektorov.
            for (int i = 0; i < pocetVektorov; i++)
```

```
            {
                vektory[i] = new Vektor3D();
                vektory[i].X = generator.Next(1, 101);
                vektory[i].Y = generator.Next(1, 101);
                vektory[i].Z = generator.Next(1, 101);
            }
            stopky.Stop();
            Console.WriteLine("Vektory sú vytvorené. Čas: [{0} ms.]",
                stopky.ElapsedMilliseconds);
            stopky.Reset();
            stopky.Start();
            Console.WriteLine("Prebiehajú matematické operácie s vektormi...");
            // Sekvenčné spracovanie matematických operácií s 3D vektormi.
            for (int i = 0; i < pocetVektorov; i++)
            {
                vektory[i].NasobitSkalarom(2); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(3); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(4); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(5); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(6); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(7); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(8); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(9); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(10); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(11); vektory[i].Normalizovat();
            }
            Console.WriteLine("Matematické operácie s vektormi sú hotové.");
            stopky.Stop();
            // Zobrazenie času spracovania sekvenčného programu.
            Console.WriteLine("\nSpracovanie sekvenčného programu " +
                "je hotové. \nČas: {0} ms.", stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu sekvenčného programu: Trojrozmerné (3D) vektory, ktoré budú podrobené matematickým operáciám, sú inštanciami triedy **Vektor3D**. V tele triedy sú definované dve trojice súkromných dátových členov, ktoré slúžia na uchovanie zložiek 3D vektora (**x**, **y**, **z**), ako aj normalizovaných zložiek 3D vektora (**norm_x**, **norm_y**, **norm_z**). Trieda **Vektor3D** ďalej obsahuje preťažený verejne prístupný inštančný konštruktor. Bezparametrická verzia inštančného konštruktora vytvára jednotkový 3D vektor. Parametrická verzia inštančného konštruktora vytvára nový

3D vektor podľa požiadaviek klienta. Funkcionalitu na realizáciu matematických operácií s 3D vektormi tvoria dve metódy:

- Preťažená inštančná metóda **NasobitSkalarom**. Táto metóda zabezpečuje násobenie 3D vektora skalárom. Bezparametrická verzia tejto metódy sama určuje skalár, ktorým bude vzápätí 3D vektor vynásobený. Parametrická verzia tejto metódy prijíma hodnotu skalára od používateľa v podobe argumentu.

- Inštančná metóda **Normalizovat**. Táto metóda vykonáva normalizáciu vektora. Akýkoľvek 3D vektor normalizujeme tak, že jeho zložky delíme veľkosťou 3D vektora. Veľkosť 3D vektora vypočítame ako druhú odmocninu zo súčtu druhých mocnín jednotlivých zložiek 3D vektora.

Okrem spomenutých metód sú v triede **Vektor3D** začlenené ešte dve verejne prístupné inštančné a bezparametrické metódy:

- Metóda **Zobrazit**: na vypísanie hodnôt zložiek 3D vektora.
- Metóda **ZobrazitVNormovanomTvare**: na vypísanie hodnôt zložiek normalizovaného 3D vektora.

Trieda **Vektor3D** umožňuje klientom zisťovanie, prípadne aj upravovanie hodnôt súkromných dátových členov (predstavujúcich zložky 3D vektora) prostredníctvom troch verejne prístupných skalárnych inštančných vlastností s identifikátormi **X**, **Y** a **Z**.

Program začína svoj život tým, že vytvorí kolekciu 10 miliónov 3D vektorov. Potom všetky 3D vektory inicializuje a začne na ne aplikovať množinu transformačných matematických operácií. Program meria čas, ktorý je nutný na:

- vytvorenie požadovanej kolekcie 3D vektorov,
- aplikáciu matematických operácií na 3D vektory.

Samozrejme, nás zaujíma iba čas, ktorý alokuje spracovanie matematických operácií s kolekciou 3D vektorov. (Čas vyžadovaný vytvorením, teda alokáciou kolekcie 3D vektorov, nemáme ako ovplyvniť, a to ani pri sekvenčnom, ani pri paralelnom programe.)

Zdrojový kód paralelného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy Vektor3D je rovnaká ako pri sekvenčnom programe.
    class Program
    {
        static void Main(string[] args)
        {
            const uint pocetVektorov = 10000000;
            Vektor3D[] vektory = new Vektor3D[pocetVektorov];
            Random generator = new Random();
            Stopwatch stopky = new Stopwatch();
            stopky.Start();
            Console.WriteLine("Prebieha vytváranie vektorov...");
            for (int i = 0; i < pocetVektorov; i++)
            {
                vektory[i] = new Vektor3D();
                vektory[i].X = generator.Next(1, 101);
                vektory[i].Y = generator.Next(1, 101);
                vektory[i].Z = generator.Next(1, 101);
            }
            stopky.Stop();
            Console.WriteLine("Vektory sú vytvorené. Čas: [{0} ms.]",
                stopky.ElapsedMilliseconds);
            stopky.Reset();
            stopky.Start();
            Console.WriteLine("Prebiehajú matematické operácie s vektormi...");
```

```
            // Paralelné spracovanie matematických operácií s 3D vektormi.
            Parallel.For(0, pocetVektorov, i =>
            {
                vektory[i].NasobitSkalarom(2); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(3); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(4); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(5); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(6); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(7); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(8); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(9); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(10); vektory[i].Normalizovat();
                vektory[i].NasobitSkalarom(11); vektory[i].Normalizovat();
            });
            Console.WriteLine("Matematické operácie s vektormi sú hotové.");
            stopky.Stop();
            Console.WriteLine("\nSpracovanie paralelného programu " +
                "je hotové. \nČas: {0} ms.", stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu paralelného programu: Paralelný program uskutočňuje paralelné spracovanie matematických operácií s alokovanou kolekciou 3D vektorov. Paralelný program implementuje techniku explicitného paralelizmu s vysokou úrovňou abstrakcie, pretože používa paralelnú platformu Parallel Extensions. Presnejšie povedané, paralelný program aplikuje dátový paralelizmus, pretože vykonáva rovnaké matematické operácie na rôznych častiach inštancie dátovej štruktúry (jednorozmerného poľa inštancií triedy **Vektor3D** v tomto prípade). Syntakticky využívame jednu z verzií preťaženej statickej parametrickej metódy **For** (rovnako statickej) triedy **Parallel** z menného priestoru **System.Threading.Tasks**.

Pre zjednodušenie budeme statickú metódu **Parallel.For** nazývať paralelným cyklom **for**. Podobne ako bežný, teda sekvenčný cyklus **for**, tak aj jeho paralelná verzia špecifikuje rozsah iterácií, ktoré budú uskutočňované. Spodná hranica paralelného cyklu **for** je inkluzívne daná prvým argumentom, ktorý je odovzdávaný statickej metóde **Parallel.For**. Horná hranica paralelného cyklu **for** je zase exkluzívne determinovaná druhým argumentom, ktorý je poskytnutý statickej metóde **Parallel.For**. Tretím argumentom statickej metódy **Parallel.For** je λ-príkaz.

Tento λ-príkaz vytvára inštanciu generického delegáta **Action<T>** (z menného priestoru **System**), definuje anonymnú metódu s množinou príkazov a odkaz na definovanú anonymnú metódu zapuzdruje do inštancie generického delegáta **Action<T>**.

Úplná deklarácia generického delegáta **Action<T>** vyzerá takto:

```
public delegate void Action<T>(T obj);
```

Ako je zrejmé, inštancia tohto delegáta môže byť spojená s (inštančnou alebo statickou) parametrickou metódou bez návratovej hodnoty. V procese inštanciácie generického delegáta **Action<T>** bude typový parameter **T** substituovaný požadovaným typovým argumentom (teda konkrétnym dátovým typom). V zdrojovom kóde paralelného programu je vytvorená inštancia delegáta **Action<long>**, ktorá je okamžite asociovaná s parametrickou anonymnou metódou.

Ako sme uviedli, celý proces je riadený nasledujúcim λ-príkazom:

```
// Paralelný cyklus riadený λ-príkazom.
Parallel.For(0, pocetVektorov, i =>
    { p₁; p₂; ...; pₙ; });
```

kde:

- **p_1**, **p_2**, ..., **p_n** sú príkazy paralelného cyklu **for**.

Na tomto mieste je potrebné poukázať na skutočnosť, že dátový typ lokálnej premennej **i** je prekladačom implicitne inferovaný. Hoci je λ-príkaz istotne intuitívnejší (pretože sprehľadňuje zdrojový kód), je pre neznalého programátora len veľmi ťažko čitateľný a zrozumiteľný. Aby sme ukázali, aký reťazec udalostí sa pri spracovaní paralelného cyklu **for** skutočne vykonáva, obmeníme syntaktický obraz predchádzajúceho paralelného cyklu **for** týmto spôsobom:

```
// Paralelný cyklus riadený explicitne inštanciovaným delegátom.
Parallel.For(0, pocetVektorov, new Action<long>(delegate(long i)
{
```

```
    p1; p2; ...; pn;
}));
```

V záujme vyhodnotenia tretieho argumentu, ktorý je poskytnutý statickej metóde **Parallel.For**, musí prekladač uskutočniť tieto aktivity:

1. Založiť inštanciu generického delegáta **Action<T>** s typovým argumentom **long**.

2. Definovať anonymnú parametrickú metódu, ktorá obsahuje jeden formálny parameter typu **long**.

3. Zapuzdriť odkaz na definovanú anonymnú parametrickú metódu do inštancie delegáta **Action<long>**.

Paralelný cyklus **for** musí aplikovať matematické operácie na množinu 10 miliónov 3D vektorov. Preto virtuálny exekučný systém segmentuje všetky iterácie paralelného cyklu **for** do zväzkov, ktoré budú vykonávané súbežne. Virtuálny exekučný systém vytvorí na pozadí adekvátny počet pracovných programových vlákien a nariadi, aby každé z týchto vlákien realizovalo spracovanie jedného zväzku paralelne vykonávaných iterácií cyklu. Pochopiteľne, paralelná implementácia programu bude korektná len vtedy, ak medzi jednotlivými iteráciami paralelného cyklu **for** nebudú žiadne interakčné väzby. Povedané inak, iterácie cyklu, ktorý má byť paralelizovaný, musia byť od seba nezávislé. Tým je zaručená úplná paralelizovateľnosť cyklu, bez nutnosti implicitnej alebo explicitnej synchronizácie.

2.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti

Sekvenčný i paralelný program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 1 a v tab. 2.

Tab. 1: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	6489	12173	9789
2.	6438	11829	9950
3.	6440	11856	9788
4.	6426	11832	9837
5.	6463	11825	9811
$\overline{E_{TS}}$	**6452**	**11903**	**9835**

Tab. 2: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	4053	6715	2913
2.	4012	6652	2937
3.	3984	6742	2974
4.	3972	6765	2976
5.	3984	6712	2920
$\overline{E_{TP}}$	**4001**	**6718**	**2944**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{6452}{4001} = \mathbf{1,61}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1{,}61}{2} \times 100 = \mathbf{80{,}5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{11903}{6718} = \mathbf{1{,}77}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800:

$$e = \frac{1{,}77}{2} \times 100 = \mathbf{88{,}5}\ \%$$

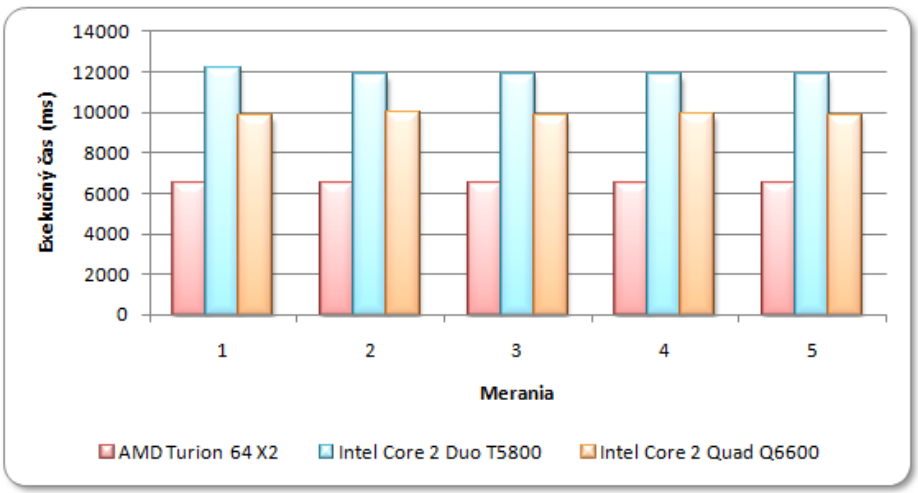
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{9835}{2944} = \mathbf{3{,}34}$$

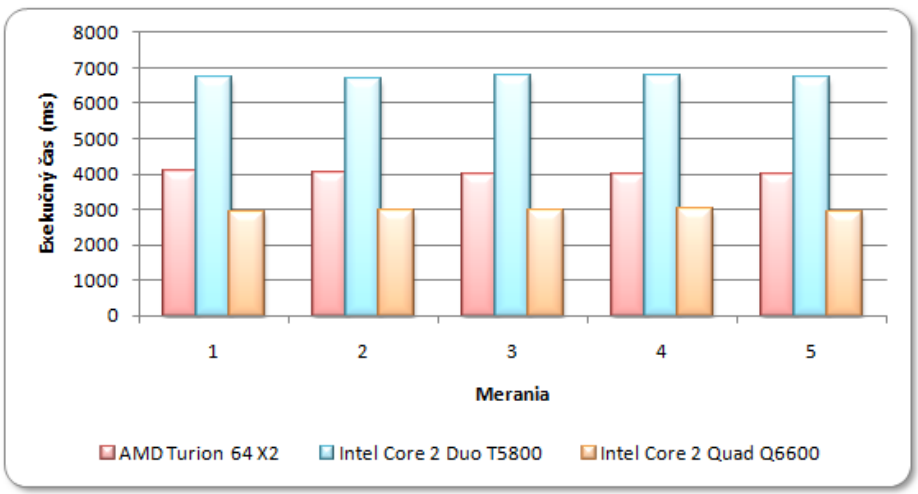
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600:

$$e = \frac{3{,}34}{4} \times 100 = \mathbf{83{,}5}\ \%$$

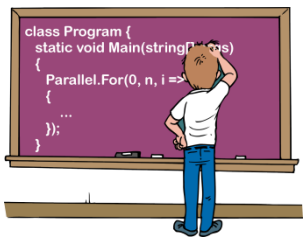


Obr. 3: Výkonnosť sekvenčného programu na testovacích počítačoch



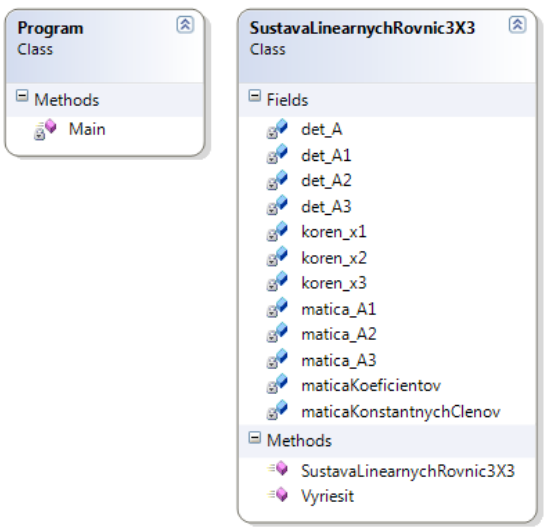
Obr. 4: Výkonnosť paralelného programu na testovacích počítačoch

3 Praktická ukážka č. 2: Riešenie masívnych súprav sústav 3 lineárnych rovníc s 3 neznámymi



Cieľ praktickej ukážky:
Paralelizácia riešenia masívnych súprav sústav 3 lineárnych rovníc s 3 neznámymi.
Vedomostná náročnosť: .
Časová náročnosť: 30 minút.
Softvérové technológie: C# 4.0 a TPL.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie: .

Vizualizácia tried sekvenčného programu:



Obr. 5: Diagram tried sekvenčného programu

Zdrojový kód sekvenčného programu:

```
using System;
using System.Collections.Generic;
```

```
using System.Diagnostics;
using System.Linq;
using System.Text;
namespace diz
{
    // Deklarácia triedy, ktorá zapuzdruje funkcionalitu na riešenie sústavy
    // 3 lineárnych rovníc s 3 neznámymi.
    class SustavaLinearnychRovnic3X3
    {
        // Definície súkromných dátových členov triedy.
        private int[,] maticaKoeficientov;
        private int[] maticaKonstantnychClenov;
        private int[,] matica_A1, matica_A2, matica_A3;
        private int det_A, det_A1, det_A2, det_A3;
        private int koren_x1, koren_x2, koren_x3;
        // Definícia parametrického inštančného konštruktora.
        public SustavaLinearnychRovnic3X3(int[,] maticaKoeficientov,
            int[] maticaKonstantnychClenov)
        {
            this.maticaKoeficientov = maticaKoeficientov;
            this.maticaKonstantnychClenov = maticaKonstantnychClenov;
            // Úprava 1. matice podľa Cramerovho pravidla.
            matica_A1 = new int[3,3];
            for(int i = 0; i < 3; i++)
            {
                for(int j = 0; j < 3; j++)
                {
                    matica_A1[i, j] = maticaKoeficientov[i, j];
                }
            }
            matica_A1[0,0] = maticaKonstantnychClenov[0];
            matica_A1[1,0] = maticaKonstantnychClenov[1];
            matica_A1[2,0] = maticaKonstantnychClenov[2];
            // Úprava 2. matice podľa Cramerovho pravidla.
            matica_A2 = new int[3,3];
            for(int i = 0; i < 3; i++)
            {
                for(int j = 0; j < 3; j++)
                {
                    matica_A2[i, j] = maticaKoeficientov[i, j];
                }
            }
            matica_A2[0,1] = maticaKonstantnychClenov[0];
            matica_A2[1,1] = maticaKonstantnychClenov[1];
            matica_A2[2,1] = maticaKonstantnychClenov[2];
            // Úprava 3. matice podľa Cramerovho pravidla.
            matica_A3 = new int[3, 3];
```

```
    for(int i = 0; i < 3; i++)
    {
        for(int j = 0; j < 3; j++)
        {
            matica_A3[i, j] = maticaKoeficientov[i, j];
        }
    }
    matica_A3[0,2] = maticaKonstantnychClenov[0];
    matica_A3[1,2] = maticaKonstantnychClenov[1];
    matica_A3[2,2] = maticaKonstantnychClenov[2];
}

// Definícia metódy na nájdenie koreňov sústavy 3 lineárnych rovníc
// s 3 neznámymi.
public void Vyriesit(bool vysledky = false)
{
    // Výpočet determinantu matice koeficientov.
    det_A = maticaKoeficientov[0,0] * maticaKoeficientov[1,1] *
            maticaKoeficientov[2,2] + maticaKoeficientov[0,1] *
            maticaKoeficientov[1,2] * maticaKoeficientov[2,0] +
            maticaKoeficientov[0,2] * maticaKoeficientov[1,0] *
            maticaKoeficientov[2,1] - maticaKoeficientov[0,2] *
            maticaKoeficientov[1,1] * maticaKoeficientov[2,0] -
            maticaKoeficientov[0,0] * maticaKoeficientov[1,2] *
            maticaKoeficientov[2,1] - maticaKoeficientov[0,1] *
            maticaKoeficientov[1,0] * maticaKoeficientov[2,2];
    // Výpočet determinantu 1. upravenej matice.
    det_A1 = matica_A1[0,0] * matica_A1[1,1] * matica_A1[2,2] +
             matica_A1[0,1] * matica_A1[1,2] * matica_A1[2,0] +
             matica_A1[0,2] * matica_A1[1,0] * matica_A1[2,1] -
             matica_A1[0,2] * matica_A1[1,1] * matica_A1[2,0] -
             matica_A1[0,0] * matica_A1[1,2] * matica_A1[2,1] -
             matica_A1[0,1] * matica_A1[1,0] * matica_A1[2,2];
    // Výpočet determinantu 2. upravenej matice.
    det_A2 = matica_A2[0,0] * matica_A2[1,1] * matica_A2[2,2] +
             matica_A2[0,1] * matica_A2[1,2] * matica_A2[2,0] +
             matica_A2[0,2] * matica_A2[1,0] * matica_A2[2,1] -
             matica_A2[0,2] * matica_A2[1,1] * matica_A2[2,0] -
             matica_A2[0,0] * matica_A2[1,2] * matica_A2[2,1] -
             matica_A2[0,1] * matica_A2[1,0] * matica_A2[2,2];
    // Výpočet determinantu 3. upravenej matice.
    det_A3 = matica_A3[0,0] * matica_A3[1,1] * matica_A3[2,2] +
             matica_A3[0,1] * matica_A3[1,2] * matica_A3[2,0] +
             matica_A3[0,2] * matica_A3[1,0] * matica_A3[2,1] -
             matica_A3[0,2] * matica_A3[1,1] * matica_A3[2,0] -
             matica_A3[0,0] * matica_A3[1,2] * matica_A3[2,1] -
             matica_A3[0,1] * matica_A3[1,0] * matica_A3[2,2];
```

```
            // Výpočet koreňov sústavy 3 lineárnych rovníc s 3 neznámymi.
            if (det_A != 0)
            {
                koren_x1 = det_A1 / det_A;
                koren_x2 = det_A2 / det_A;
                koren_x3 = det_A3 / det_A;
            }
            // Ak je aktivovaná príslušná voľba, program vypíše determinanty
            // a množinu koreňov.
            if (vysledky)
            {
                Console.WriteLine(" det_A = {0}.", det_A);
                Console.WriteLine("det_A1 = {0}.", det_A1);
                Console.WriteLine("det_A2 = {0}.", det_A2);
                Console.WriteLine("det_A3 = {0}.", det_A3);
                Console.WriteLine("\nK = {{{0}, {1}, {2}}}.",
                    koren_x1, koren_x2, koren_x3);
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            SustavaLinearnychRovnic3X3[] sustavy;
            Random generator = new Random();
            const uint pocetSustav = 1000000;
            sustavy = new SustavaLinearnychRovnic3X3[pocetSustav];
            Stopwatch stopky = new Stopwatch();
            Console.WriteLine("Vytvára sa súprava {0} sústav " +
                "lineárnych rovníc.", pocetSustav);
            stopky.Start();
            // Vytvorenie súpravy sústav lineárnych rovníc.
            for (int i = 0; i < pocetSustav; i++)
            {
                sustavy[i] = new SustavaLinearnychRovnic3X3(
                    new int[,]
                    {
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)},
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)},
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)}
                    },
                    new int[]
                    {
```

```
                    generator.Next(1, 11), generator.Next(1, 11),
                    generator.Next(1, 11)
                });
            }
            stopky.Stop();
            Console.WriteLine("Súprava lineárnych rovníc je vytvorená " +
                "[čas: {0} ms].", stopky.ElapsedMilliseconds);
            stopky.Reset();
            Console.WriteLine("Začalo sa riešenie súpravy {0} sústav " +
                "lineárnych rovníc.", pocetSustav);
            stopky.Start();

            // Sekvenčné riešenie súpravy sústav lineárnych rovníc.
            for (int i = 0; i < pocetSustav; i++)
            {
                for (int j = 1; j <= 100; j++)
                {
                    sustavy[i].Vyriesit();
                }
            }
            stopky.Stop();
            Console.WriteLine("Sekvenčné riešenie súpravy sústav " +
                "lineárnych rovníc je hotové [čas: {0} ms].",
                stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu sekvenčného programu: Všetku funkcionalitu, ktorá sa viaže na riešenie sústavy 3 lineárnych rovníc s 3 neznámymi, sme umiestnili do triedy **SustavaLinearnychRovnic3X3**. V súkromnej sekcii triedy definujeme dátové členy, ktoré budeme potrebovať. Ide o maticu koeficientov, maticu konštantných členov pravej strany, trojicu matíc, ktoré budú upravované podľa Cramerovho pravidla, determinanty a korene sústavy lineárnych rovníc. Parametrický inštančný konštruktor triedy očakáva, že klientsky kód mu poskytne odkazy na maticu koeficientov a maticu konštantných členov pravej strany. Ako si môžeme všimnúť, matica koeficientov je syntakticky reprezentovaná dvojrozmerným (2D) celočíselným poľom. Keďže matica konštantných členov pravej strany je vektorom, stačí nám na jej uchovanie jednorozmerné (1D) pole. Okrem toho, že v tele konštruktora získame prístup k požadovaným vstupným maticiam, vytvárame

ďalšie tri matice, ktoré inicializujeme podľa Cramerovho pravidla. To znamená, že i-tý stĺpec každej z týchto troch matíc je nahradený vektorom konštantných členov pravej strany sústavy.

Sústavu 3 lineárnych rovníc s 3 neznámymi rieši verejne prístupná, inštančná a bezparametrická metóda **Vyriesit**. Aby mohla metóda nájsť množinu koreňov, musí najskôr vypočítať determinant matice koeficientov a rovnako aj determinanty troch matíc modifikovaných podľa Cramerovho pravidla. V momente, keď sú hodnoty všetkých determinantov známe, metóda vypočíta korene sústavy a spoločne s determinantmi ich zobrazí na výstupe.

V tele hlavnej metódy **Main** konštruujeme súpravu, ktorá obsahuje 1 milión sústav 3 lineárnych rovníc s 3 neznámymi. Len čo sú všetky sústavy lineárnych rovníc správne inicializované, štartujeme proces ich riešenia. Aby sme zvýšili zaťaženie systému, každú sústavu lineárnych rovníc riešime opakovane 100-krát.

Zdrojový kód paralelného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy SustavaLinearnychRovnic3X3 je rovnaká
    // ako pri sekvenčnom programe.
    class Program
    {
        static void Main(string[] args)
        {
            SustavaLinearnychRovnic3X3[] sustavy;
            Random generator = new Random();
            const uint pocetSustav = 1000000;
            sustavy = new SustavaLinearnychRovnic3X3[pocetSustav];
            Stopwatch stopky = new Stopwatch();
            Console.WriteLine("Vytvára sa súprava {0} sústav " +
```

```
                "lineárnych rovníc.", pocetSustav);
            stopky.Start();
            for (int i = 0; i < pocetSustav; i++)
            {
                sustavy[i] = new SustavaLinearnychRovnic3X3(
                    new int[,]
                    {
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)},
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)},
                        {generator.Next(1, 11), generator.Next(1, 11),
                         generator.Next(1, 11)}
                    },
                    new int[]
                    {
                        generator.Next(1, 11), generator.Next(1, 11),
                        generator.Next(1, 11)
                    });
            }
            stopky.Stop();
            Console.WriteLine("Súprava lineárnych rovníc je vytvorená " +
                "[čas: {0} ms].", stopky.ElapsedMilliseconds);
            stopky.Reset();
            Console.WriteLine("Začalo sa riešenie súpravy {0} sústav " +
                "lineárnych rovníc.", pocetSustav);
            stopky.Start();

            // Paralelné riešenie súpravy sústav lineárnych rovníc.
            Parallel.For(0, pocetSustav, i =>
            {
                for (int j = 0; j <= 100; j++)
                {
                    sustavy[i].Vyriesit();
                }
            });
            stopky.Stop();
            Console.WriteLine("Paralelné riešenie súpravy sústav " +
                "lineárnych rovníc je hotové [čas: {0} ms].",
                stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu paralelného programu: Paralelný program uplatňuje techniku explicitného dátového paralelizmu s vysokou úrovňou abstrakcie. Paralelné riešenie

súpravy sústav 3 lineárnych rovníc s 3 neznámymi vykonáva paralelný cyklus **for**, ktorý je riadený λ-príkazom.

3.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti

Sekvenčný i paralelný program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 3 a v tab. 4.

Tab. 3: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	39039	17689	14805
2.	39080	17699	14722
3.	39118	17690	14894
4.	39084	17684	14786
5.	39058	17734	14721
$\overline{E_{TS}}$	**39076**	**17700**	**14786**

Tab. 4: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	19891	9483	3898
2.	19782	9272	3918
3.	19715	9323	4122
4.	19810	9195	4124
5.	19777	9184	3957
$\overline{E_{TP}}$	**19795**	**9292**	**4004**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{39076}{19795} = \mathbf{1,97}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1,97}{2} \times 100 = \mathbf{98,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{17700}{9292} = \mathbf{1,90}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}90}{2} \times 100 = \mathbf{95}\ \%$$

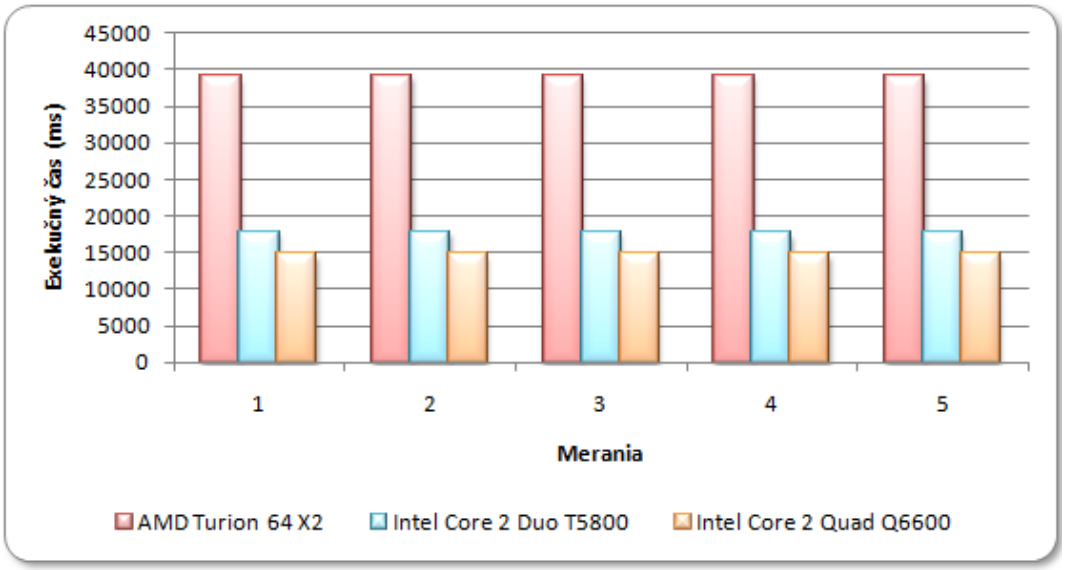
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{14786}{4004} = \mathbf{3{,}69}$$

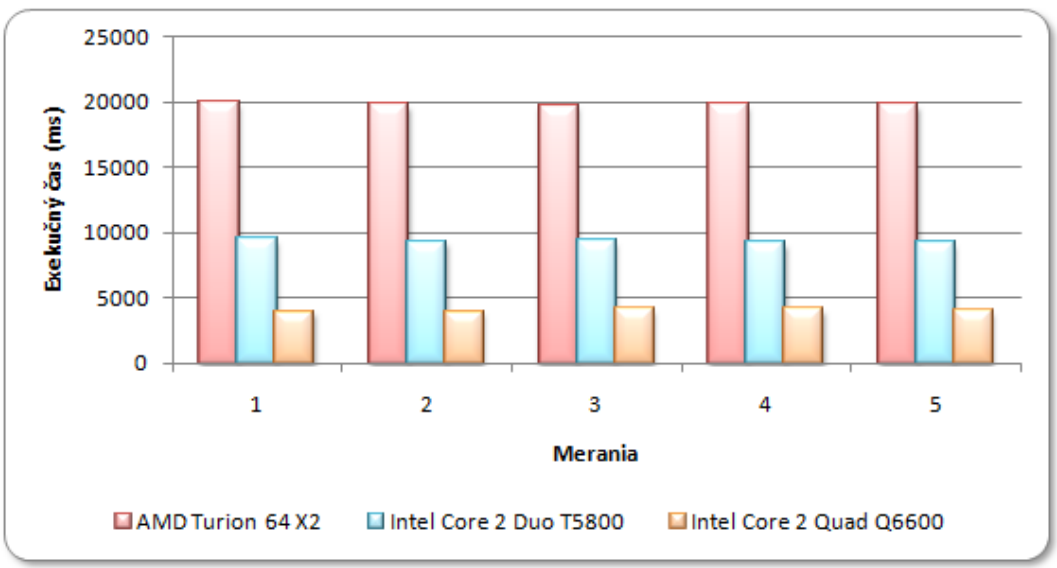
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}69}{4} \times 100 = \mathbf{92{,}25}\ \%$$

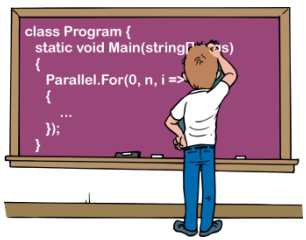


Obr. 6: Výkonnosť sekvenčného programu na testovacích počítačoch



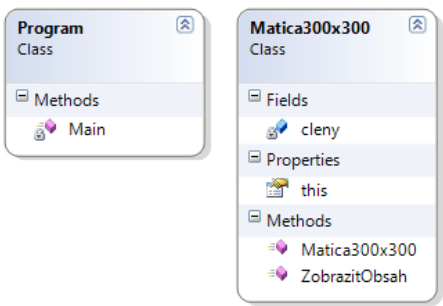
Obr. 7: Výkonnosť paralelného programu na testovacích počítačoch

4 Praktická ukážka č. 3: Lineárna algebra – Násobenie matíc typu 300x300



Cieľ praktickej ukážky:
Paralelizácia spracovania súčinu matíc typu 300 x 300.
Vedomostná náročnosť: .
Časová náročnosť: 15 minút.
Softvérové technológie: C# 4.0 a TPL.
Druh paralelizmu: Explicitný úlohový a dátový paralelizmus.
Úroveň abstrakcie: .

Vizualizácia tried sekvenčného programu:



Obr. 8: Diagram tried sekvenčného programu

Zdrojový kód sekvenčného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy, ktorej inštancie budú maticami typu 300x300.
    class Matica300x300
```

```
{
    // Definícia dátového člena triedy, ktorý bude inicializovaný
    // odkazom na dvojrozmerné pole s dátami matice.
    private int[,] cleny;
    // Definícia verejného bezparametrického inštančného konštruktora.
    public Matica300x300()
    {
        // Inicializácia matice typu 300x300.
        cleny = new int[300, 300];
        for (int i = 0; i < 300; i++)
        {
            for (int j = 0; j < 300; j++)
            {
                cleny[i, j] = i + j + 1;
            }
        }
    }
    // Definícia verejnej parametrickej indexovanej inštančnej vlastnosti.
    public int this[int i, int j]
    {
        get
        {
            return cleny[i, j];
        }
        set
        {
            cleny[i, j] = value;
        }
    }
    // Definícia verejnej bezparametrickej inštančnej metódy
    // na zobrazenie obsahu matice.
    public void ZobrazitObsah()
    {
        for (int i = 0; i < 300; i++)
        {
            for (int j = 0; j < 300; j++)
            {
                Console.Write("{0} ", cleny[i, j]);
            }
            Console.WriteLine();
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
```

```
const int pocetMatic = 40;
// Vytvorenie 6 vektorových polí.
// V každom poli sa bude nachádzať 40 matíc typu 300x300.
Matica300x300[] m1 = new Matica300x300[pocetMatic];
Matica300x300[] m2 = new Matica300x300[pocetMatic];
Matica300x300[] m3 = new Matica300x300[pocetMatic];
Matica300x300[] m4 = new Matica300x300[pocetMatic];
Matica300x300[] m5 = new Matica300x300[pocetMatic];
Matica300x300[] m6 = new Matica300x300[pocetMatic];
// Inicializácia vektorových polí matíc.
for (int i = 0; i < pocetMatic; i++)
{
    m1[i] = new Matica300x300();
    m2[i] = new Matica300x300();
    m3[i] = new Matica300x300();
    m4[i] = new Matica300x300();
    m5[i] = new Matica300x300();
    m6[i] = new Matica300x300();
}
Stopwatch stopky = new Stopwatch();
stopky.Start();
// Sekvenčné násobenie dvojice matíc typu 300x300.
Console.WriteLine("Sekvenčné násobenie matíc.");
Console.WriteLine("Úloha 1: Násobenie matíc.");
for (int i = 0; i < pocetMatic; i++)
{
    for (int a = 0; a < 300; a++)
    {
        for (int b = 0; b < 300; b++)
        {
            m5[i][a, b] = 0;
            for (int k = 0; k < 300; k++)
            {
                m5[i][a, b] += m1[i][a, k] * m2[i][k, b];
            }
        }
    }
}

// Sekvenčné násobenie ďalšej dvojice matíc typu 300x300.
Console.WriteLine("Úloha 2: Násobenie matíc.");
for (int i = 0; i < pocetMatic; i++)
{
    for (int a = 0; a < 300; a++)
    {
        for (int b = 0; b < 300; b++)
        {
```

```
                        m6[i][a, b] = 0;
                        for (int k = 0; k < 300; k++)
                        {
                            m6[i][a, b] += m3[i][a, k] * m4[i][k, b];
                        }
                    }
                }
            }
            stopky.Stop();
            Console.WriteLine("Celkový čas trvania: {0} ms.",
                stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu sekvenčného programu: Inštancie triedy **Matica300x300** budú štvorcovými maticami, ktoré použijeme pri multiplikatívnych operáciách. V deklarácii triedy **Matica300x300** sa objavuje verejný bezparametrický inštančný konštruktor, ktorý zabezpečuje inicializáciu každej založenej inštancie tejto triedy. V záujme poskytnutia elegantného prístupu k prvkom matice sme do tela triedy zapracovali takisto definíciu verejnej parametrickej indexovanej inštančnej vlastnosti. V signatúre tejto vlastnosti sa vyskytujú 2 formálne parametre, pomocou ktorých možno priamo, pomocou indexovej notácie, identifikovať prvok matice, ku ktorému chceme pristupovať.

V tele hlavnej metódy **Main** vytvárame 6 40-prvkových vektorových polí s tým, že v každom poli sa nachádza 40 inštancií triedy **Matica300x300** (teda 40 štvorcových matíc). Vo chvíli, keď sú matice inicializované, vypočítavame ich súčin. Čas, ktorý je potrebný na spracovanie multiplikatívnej operácie, zaznamenávame a nakoniec zobrazujeme na výstupe.

Zdrojový kód paralelného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
```

```
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy Matica300x300 je rovnaká
    // ako pri sekvenčnom programe.
    class Program
    {
        static void Main(string[] args)
        {
            const int pocetMatic = 40;
            Matica300x300[] m1 = new Matica300x300[pocetMatic];
            Matica300x300[] m2 = new Matica300x300[pocetMatic];
            Matica300x300[] m3 = new Matica300x300[pocetMatic];
            Matica300x300[] m4 = new Matica300x300[pocetMatic];
            Matica300x300[] m5 = new Matica300x300[pocetMatic];
            Matica300x300[] m6 = new Matica300x300[pocetMatic];

            for (int i = 0; i < pocetMatic; i++)
            {
                m1[i] = new Matica300x300();
                m2[i] = new Matica300x300();
                m3[i] = new Matica300x300();
                m4[i] = new Matica300x300();
                m5[i] = new Matica300x300();
                m6[i] = new Matica300x300();
            }

            Stopwatch stopky = new Stopwatch();
            stopky.Start();

            Console.WriteLine("Paralelné násobenie matíc.");
            // Vytvorenie úlohy, ktorá bude realizovať paralelné násobenie
            // prvej dvojice matíc typu 300x300.
            Task uloha1 = new Task(() =>
            {
                Console.WriteLine("Úloha 1: Násobenie matíc.");
                Parallel.For(0, pocetMatic, i =>
                {
                    for (int a = 0; a < 300; a++)
                    {
                        for (int b = 0; b < 300; b++)
                        {
                            m5[i][a, b] = 0;
                            for (int k = 0; k < 300; k++)
                            {
```

```
                            m5[i][a, b] += m1[i][a, k] * m2[i][k, b];
                        }
                    }
                }
            });
        });

        // Vytvorenie úlohy, ktorá bude realizovať paralelné násobenie
        // druhej dvojice matíc typu 300x300.
        Task uloha2 = new Task(() =>
        {
            Console.WriteLine("Úloha 2: Násobenie matíc.");
            Parallel.For(0, pocetMatic, i =>
            {
                for (int a = 0; a < 300; a++)
                {
                    for (int b = 0; b < 300; b++)
                    {
                        m6[i][a, b] = 0;
                        for (int k = 0; k < 300; k++)
                        {
                            m6[i][a, b] += m3[i][a, k] * m4[i][k, b];
                        }
                    }
                }
            });
        });
        // Začatie spracovania úloh.
        uloha1.Start();
        uloha2.Start();
        // Primárne programové vlákno čaká, až dokiaľ nebudú hotové
        // všetky paralelne uskutočňované úlohy.
        Task.WaitAll(uloha1, uloha2);

        stopky.Stop();
        Console.WriteLine("Celkový čas trvania: {0} ms.",
            stopky.ElapsedMilliseconds);
    }
  }
}
```

Verbálny súhrn zdrojového kódu paralelného programu: Paralelná verzia programu na výpočet súčinu štvorcových matíc typu 300 x 300 implementuje techniku kombinovaného (úlohového

a dátového) paralelizmu. Filozofia, ktorá stojí za návrhom paralelne pracujúceho zdrojového kódu programu, je nasledujúca:

- Každú multiplikatívnu operáciu zapuzdrime do jednej úlohy. V tomto kontexte termín úloha predstavuje výpočtový proces, ktorý môže byť súbežne realizovaný s inými výpočtovými procesmi (s inými úlohami). Keďže chceme paralelne spracovať súčiny 2 dvojíc štvorcových matíc, dokážeme súčin každej dvojice matíc zapuzdriť do jednej diskrétnej úlohy, ktorá je schopná paralelného spracovania.

- Vo všeobecnosti sa pri paralelizácii existujúceho sekvenčného programu snažíme diagnostikovať diskrétne výpočtové procesy, ktoré umožňujú súbežnú exekúciu. Optimálne je, keď medzi takto identifikovanými úlohami neexistujú žiadne explicitné dátové, prípadne komunikačné väzby. Hoci táto praktická ukážka je naprojektovaná tak, aby toto optimalizačné kritérium spĺňala, vieme, že v praxi sa často vyskytujú situácie, kedy sa medzi úlohami vyskytujú vzájomné interakčné vzťahy. Potom je potrebné tieto väzby dôkladne analyzovať a navrhnúť efektívny model implementácie úlohového paralelizmu s uvážlivým výberom synchronizačných primitív, ktoré budú v determinovaných časových intervaloch riadiť komunikačné toky medzi úlohami.

- Z technického hľadiska je úloha v zdrojovom kóde programovacieho jazyka C# 4.0 reprezentovaná inštanciou triedy **Task** z menného priestoru **System.Threading.Tasks**. Parametrický inštančný konštruktor triedy **Task** prijíma odkaz na inštanciu delegáta **Action**. Inštanciu tohto delegáta a rovnako aj anonymnú metódu, ktorá bude s inštanciou delegáta asociovaná, vytvárame pomocou príslušného λ-výrazu. V tele λ-výrazu sa nachádza programový kód, ktorý vykonáva násobenie 2 štvorcových matíc typu 300 x 300 z 2 vektorových polí, a ktorý inicializuje výslednú súčinovú maticu ďalšieho vektorového poľa. Pôvodne sekvenčný vonkajší cyklus **for** sme nahradili jeho paralelným variantom. Týmto spôsobom vytvárame úlohu, ktorá implementuje dátový paralelizmus. Vzhľadom na to, že

jednotlivé iterácie paralelného cyklu **for** sú od seba nezávislé, môžu byť explicitne paralelizované. Na identickom pracovnom modeli je postavený aj pracovný model druhej úlohy, ktorá vykonáva súčin 2 štvorcových matíc ostatných vektorových polí.

- Plánovač úloh garantuje uskutočnenie mapovania úloh na pracovné programové vlákna. Vo všeobecnosti sa odporúča, aby programátori pracovali namiesto vlákien s úlohami, pretože tie poskytujú vyššiu mieru abstrakcie pri realizácii paralelných operácií. Skutočnosti, koľko pracovných vlákien bude vytvorených, ako budú na pracovné vlákna mapované úlohy, ako bude do pracovných vlákien injektovaný preložený kód úloh, a ako budú riadené životné cykly pracovných vlákien, nemusia vývojárov zaujímať. O všetky nízkoúrovňové technické detaily sa postará plánovač úloh a virtuálny exekučný systém v kooperácii s operačným systémom a hardvérovou platformou počítačovej stanice, na ktorej paralelný program beží. Pracovné vlákna žijú minimálne tak dlho, ako trvá spracovanie úloh, ktoré boli na tieto vlákna delegované. V praxi však pracovné vlákna žijú dlhšie, pričom po spracovaní jednej série úloh začínajú realizovať ďalšiu sériu úloh. Takýto prístup je efektívny, a to z dvoch dôvodov. Po prvé, plánovač úloh dbá na to, aby boli všetky existujúce pracovné vlákna zaneprázdnené realizáciou programových operácií. Po druhé, zámerom plánovača úloh je maximalizácia opätovného využitia existujúcich pracovných vlákien paralelného programu. Podotknime, že vytvorenie pracovných vlákien je akcia náročná na čas a výkon. Preto sa plánovač úloh snaží už vytvorené vlákna v čo možno najväčšej miere recyklovať. To má jednoznačne blahodarný vplyv na exekučnú efektívnosť paralelného programu.

- Inštanciáciou triedy **Task** však úlohy nie sú ihneď naplánované na explicitné spracovanie. To sa deje až po aktivácii inštančnej metódy **Start**. Rozhranie triedy **Task** tvorí niekoľko statických metód, pomocou ktorých môžeme s úlohami vykonávať rôzne operácie. V predstavenom zdrojovom kóde voláme statickú metódu **WaitAll**, ktorá pozastaví spracovanie kódu na

primárnom vlákne až dovtedy, kým nebudú spracované všetky špecifikované úlohy.

4.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti

Sekvenčný i paralelný program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 5 a v tab. 6.

Tab. 5: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	72946	44535	36691
2.	72791	44151	36701
3.	72673	44139	36656
4.	72839	43802	36679
5.	72813	43860	36798
$\overline{E_{TS}}$	**72813**	**44098**	**36705**

Tab. 6: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	40997	23707	10176
2.	40754	23504	10198
3.	41764	23628	10247
4.	40903	23989	10225
5.	40752	23602	10228
$\overline{E_{TP}}$	**41034**	**23686**	**10215**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{72813}{41034} = \mathbf{1,77}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1,77}{2} \times 100 = \mathbf{88,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{44098}{23686} = \mathbf{1,86}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}86}{2} \times 100 = \mathbf{93}\ \%$$

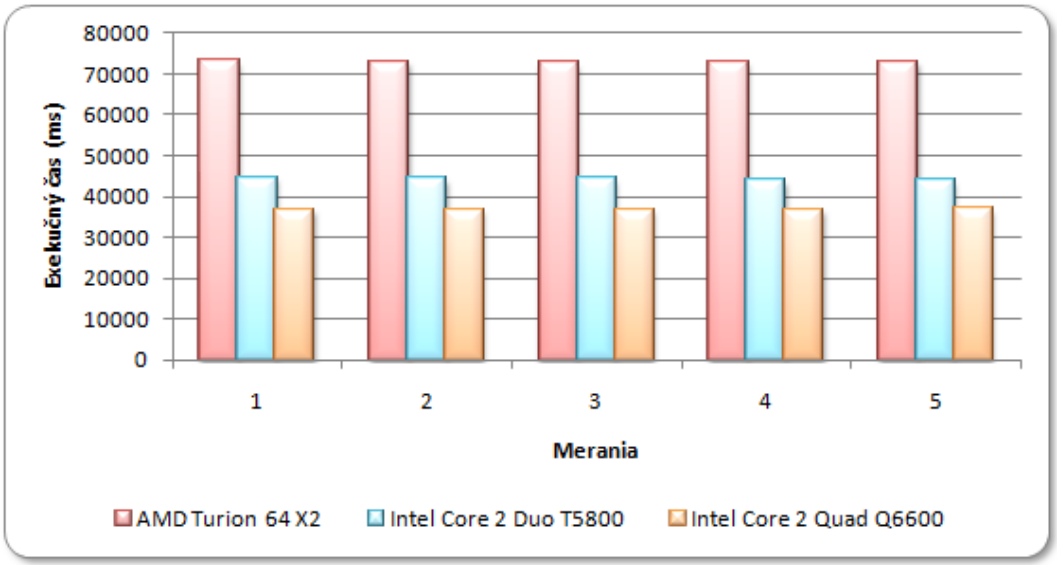
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{36705}{10215} = \mathbf{3{,}59}$$

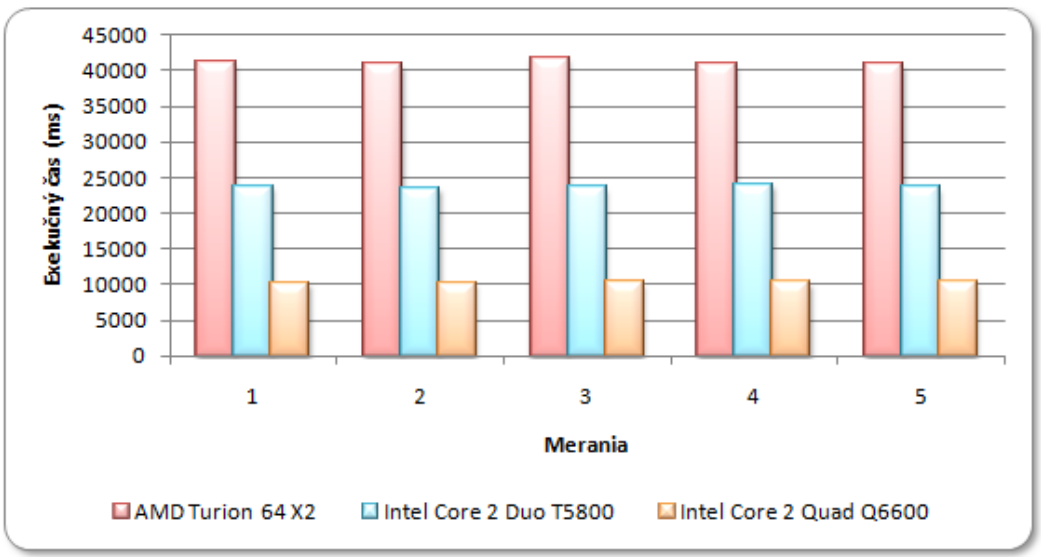
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}59}{4} \times 100 = \mathbf{89{,}75}\ \%$$

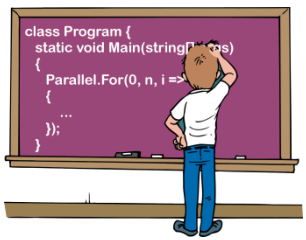


Obr. 9: Výkonnosť sekvenčného programu na testovacích počítačoch



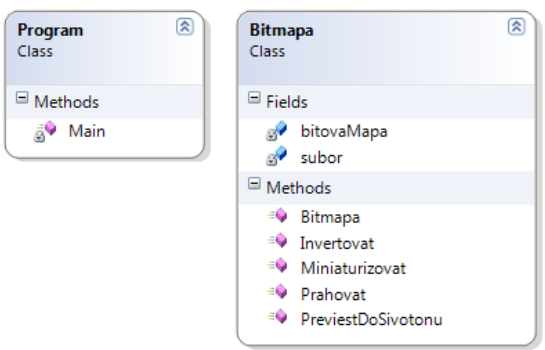
Obr. 10: Výkonnosť paralelného programu na testovacích počítačoch

5 Praktická ukážka č. 4: Paralelné grafické transformácie bitových máp (paralelizmus s vysokou úrovňou abstrakcie)



Cieľ praktickej ukážky:
Paralelizácia grafických transformácií bitových máp.
Vedomostná náročnosť:  .
Časová náročnosť: 30 minút.
Softvérové technológie: C# 4.0 a TPL.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie:  .

Vizualizácia tried sekvenčného programu:



Obr. 11: Diagram tried sekvenčného programu

Zdrojový kód sekvenčného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;

namespace diz
{
```

```
// Deklarácia triedy, ktorá zapuzdruje funkcionalitu na realizáciu
// vybraných grafických transformácií s bitovými mapami.
class Bitmapa
{
    // Definície súkromných dátových členov.
    private Bitmap bitovaMapa;
    private string subor;
    // Verejne prístupný parametrický inštančný konštruktor.
    public Bitmapa(string subor)
    {
        bitovaMapa = new Bitmap(subor);
        this.subor = subor;
    }
    // Metóda realizujúca inverziu (tvorbu negatívu) bitovej mapy.
    public void Invertovat()
    {
        int x, y;
        Color farba;
        for (x = 0; x < bitovaMapa.Width; x++)
        {
            for (y = 0; y < bitovaMapa.Height; y++)
            {
                farba = bitovaMapa.GetPixel(x, y);
                bitovaMapa.SetPixel(x, y,
                    Color.FromArgb(255 - farba.R, 255 - farba.G,
                    255 - farba.B));
            }
        }
        FileInfo iSubor = new FileInfo(this.subor);
        string aktualnyPriecinok = iSubor.DirectoryName;
        string novyPriecinok = Directory.CreateDirectory(aktualnyPriecinok +
            "\\IMG_I").FullName;
        bitovaMapa.Save(Path.Combine(novyPriecinok, iSubor.Name));
    }
    // Metóda realizujúca prevod bitovej mapy do sivotónu.
    public void PreviestDoSivotonu()
    {
        int x, y, I;
        Color farba;
        for (x = 0; x < bitovaMapa.Width; x++)
        {
            for (y = 0; y < bitovaMapa.Height; y++)
            {
                farba = bitovaMapa.GetPixel(x, y);
                I = (int)(0.299 * farba.R + 0.587 * farba.G +
                    0.114 * farba.B);
                bitovaMapa.SetPixel(x, y, Color.FromArgb(I, I, I));
```

```
            }
        }
        FileInfo iSubor = new FileInfo(this.subor);
        string aktualnyPriecinok = iSubor.DirectoryName;
        string novyPriecinok = Directory.CreateDirectory(aktualnyPriecinok +
            "\\IMG_S").FullName;
        bitovaMapa.Save(Path.Combine(novyPriecinok, iSubor.Name));
    }
    // Metóda realizujúca prahovanie bitovej mapy.
    public void Prahovat(int prah)
    {
        int x, y, cervena, zelena, modra;
        Color farba;
        for (x = 0; x < bitovaMapa.Width; x++)
        {
            for (y = 0; y < bitovaMapa.Height; y++)
            {
                farba = bitovaMapa.GetPixel(x, y);
                if (farba.R <= prah)
                    cervena = 0;
                else
                    cervena = 255;
                if (farba.G <= prah)
                    zelena = 0;
                else
                    zelena = 255;
                if (farba.B <= prah)
                    modra = 0;
                else
                    modra = 255;
                bitovaMapa.SetPixel(x, y,
                    Color.FromArgb(cervena, zelena, modra));
            }
        }
        FileInfo iSubor = new FileInfo(this.subor);
        string aktualnyPriecinok = iSubor.DirectoryName;
        string novyPriecinok = Directory.CreateDirectory(aktualnyPriecinok +
            "\\IMG_P").FullName;
        bitovaMapa.Save(Path.Combine(novyPriecinok, iSubor.Name));
    }
    // Metóda vytvárajúca miniatúru bitovej mapy.
    public void Miniaturizovat()
    {
        FileInfo iSubor = new FileInfo(this.subor);
        string aktualnyPriecinok = iSubor.DirectoryName;
        string novyPriecinok = Directory.CreateDirectory(aktualnyPriecinok +
            "\\IMG_Mini").FullName;
```

```
            (bitovaMapa.GetThumbnailImage(bitovaMapa.Width / 10,
                bitovaMapa.Height / 10, null,
                IntPtr.Zero)).Save(Path.Combine(novyPriecinok, iSubor.Name));
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DirectoryInfo priecinok = new DirectoryInfo(@"c:\Obrazky");
            FileInfo[] subory = priecinok.GetFiles("*.jpg");
            ushort pocetSuborov = (ushort)subory.Length;
            Bitmapa[] bitmapy = new Bitmapa[pocetSuborov];
            Stopwatch stopky = new Stopwatch();
            // Vytvorenie kolekcie bitových máp.
            for(int i = 0; i < pocetSuborov; i++)
            {
                bitmapy[i] = new Bitmapa(subory[i].FullName);
            }
            // Sekvenčné spracovanie grafických transformácií bitových máp.
            Console.WriteLine("Prebiehajú sekvenčné grafické " +
                "transformácie bitových máp.");
            stopky.Start();
            for (int i = 0; i < pocetSuborov; i++)
            {
                bitmapy[i].Invertovat();
                bitmapy[i].PreviestDoSivotonu();
                bitmapy[i].Prahovat(128);
            }
            stopky.Stop();
            Console.WriteLine("Sekvenčné grafické transformácie " +
                "sú hotové [čas: {0} ms].", stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu sekvenčného programu: Na inštanciách triedy **Bitmapa** môžu byť uskutočňované tri typy grafických transformácií:

1. **Inverzia (tvorba negatívu) bitovej mapy.** Podstatou inverzie bitovej mapy je invertovanie farebných vektorov všetkých obrazových bodov, z ktorých je bitová mapa zložená. Ak *P* je farebný vektor obrazového bodu so zložkami R, G, B (pričom tieto zložky môžu nadobúdať celočíselné hodnoty z intervalu

<0, 255>), tak invertovaný farebný vektor P' získame prostredníctvom tohto matematického vzorca: $P'[255 - R, 255 - G, 255 - B]$.

2. **Prevod bitovej mapy do sivotónu.** Pri prevode bitovej mapy do sivotónu najskôr vypočítame hodnotu jasu (I), ktorá zodpovedá zobrazeniu farieb v sivotóne. Na to použijeme nasledujúci matematický vzťah:

$$I = 0{,}299 \times R + 0{,}587 \times G + 0{,}114 \times B$$

kde:

- R, G, B sú zložky farebného vektora obrazového bodu.

Farebný vektor obrazového bodu, ktorý bol prevedený do sivotónu, bude mať túto podobu: $P'[I, I, I]$.

3. **Prahovanie bitovej mapy.** Pri prahovaní bitovej mapy postupujeme tak, že najskôr určíme tzv. prahovú hodnotu (p). Prahová hodnota bude v našom prípade celočíselná hodnota z intervalu <0, 255>. Algoritmus grafickej transformácie porovnáva zložky farebného vektora R, G a B obrazového bodu P s prahovou hodnotou. Pritom sa uplatňuje táto súprava pravidiel:

$$R \leq p \Rightarrow R = 0$$
$$R > p \Rightarrow R = 255$$

$$G \leq p \Rightarrow G = 0$$
$$G > p \Rightarrow G = 255$$

$$B \leq p \Rightarrow B = 0$$
$$B > p \Rightarrow B = 255$$

Algoritmus grafickej transformácie zisťuje, či je zložka farebného vektora obrazového bodu menšia alebo rovná prahovej hodnote. Ak áno, dochádza k nastaveniu zložky farebného vektora na minimálnu hodnotu (0). Ak je

zložka farebného vektora obrazového bodu väčšia ako prahová hodnota, tak bude nastavená na maximálnu hodnotu (255).

Na obr. 12 sú znázornené výsledné efekty grafických transformácií.

Obr. 12: Postupne zľava doprava: originálna bitmapa, invertovaná bitmapa, bitmapa v sivotóne a bitmapa po prahovaní (s prahovou hodnotou 128).

Sekvenčný program najskôr otvorí zadaný priečinok (v našom prípade ide o priečinok Obrazky, ktorý sa nachádza na disku C), a potom načíta všetky bitové mapy (s príponou .jpg), ktoré sa v tomto priečinku nachádzajú. V ďalšom kroku program uskutoční s bitovými mapami špecifikované grafické transformácie. Nové bitové mapy, ktoré vzniknú ako produkty algoritmov grafických transformácií, budú uskladnené v samostatných priečinkoch (tieto novo vytvorené priečinky budú pôsobiť ako vnorené priečinky hlavného priečinka Obrazky).

Zdrojový kód paralelného programu:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy Bitmapa je rovnaká ako pri sekvenčnom programe.
    class Program
    {
        static void Main(string[] args)
        {
```

```
            DirectoryInfo priecinok = new DirectoryInfo(@"c:\Obrazky");
            FileInfo[] subory = priecinok.GetFiles("*.jpg");
            ushort pocetSuborov = (ushort)subory.Length;
            Bitmapa[] bitmapy = new Bitmapa[pocetSuborov];
            Stopwatch stopky = new Stopwatch();
            for (int i = 0; i < pocetSuborov; i++)
            {
                bitmapy[i] = new Bitmapa(subory[i].FullName);
            }
            Console.WriteLine("Prebiehajú paralelné grafické transformácie " +
                "bitových máp.");
            stopky.Start();
            // Paralelné spracovanie grafických transformácií bitových máp
            // pomocou paralelného cyklu foreach, ktorý je riadený λ-príkazom.
            Parallel.ForEach(bitmapy, bitmapa =>
            {
                bitmapa.Invertovat();
                bitmapa.PreviestDoSivotonu();
                bitmapa.Prahovat(128);
            });
            stopky.Stop();
            Console.WriteLine("Paralelné grafické transformácie sú " +
                "hotové [čas: {0} ms].", stopky.ElapsedMilliseconds);
        }
    }
}
```

Verbálny súhrn zdrojového kódu paralelného programu: Paralelný program uplatňuje techniku explicitného dátového paralelizmu s vysokou úrovňou abstrakcie. Paralelné spracovanie grafických transformácií bitových máp realizuje paralelný cyklus **foreach**, ktorý je riadený λ-príkazom.

5.1 Empirické testovanie sekvenčného a paralelného programu a kvantifikácia nárastu výkonnosti

Sekvenčný i paralelný program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 7 a v tab. 8.

Tab. 7: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	125101	83148	66081
2.	124108	84530	65738
3.	123957	84508	65465
4.	123941	84507	66589
5.	124239	84576	66265
$\overline{E_{TS}}$	**124270**	**84254**	**66028**

Tab. 8: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	64846	44436	19010
2.	60668	43877	18383
3.	60798	43835	18842
4.	61028	44028	19235
5.	62931	43602	18253
$\overline{E_{TP}}$	**62055**	**43956**	**18745**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{124270}{62055} = \mathbf{2,00}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2 je **lineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{2{,}00}{2} \times 100 = \mathbf{100}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{84254}{43956} = \mathbf{1{,}92}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}92}{2} \times 100 = \mathbf{96}\ \%$$

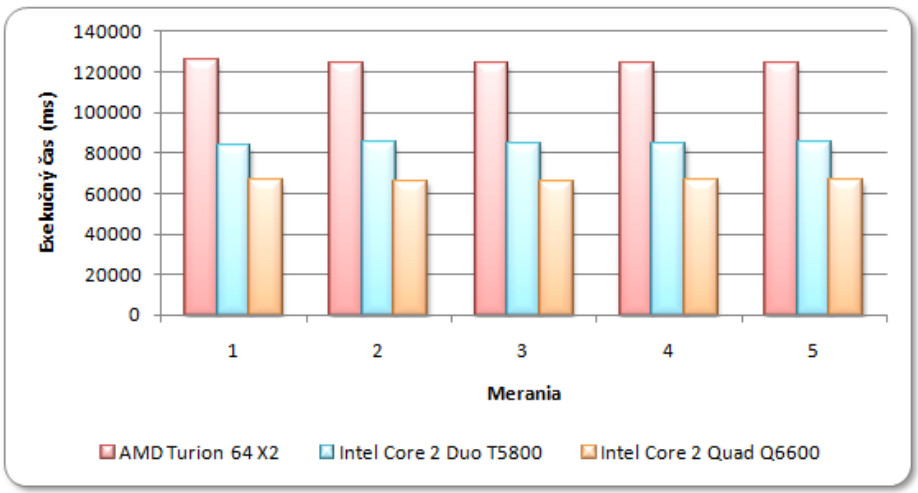
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{66028}{18745} = \mathbf{3{,}52}$$

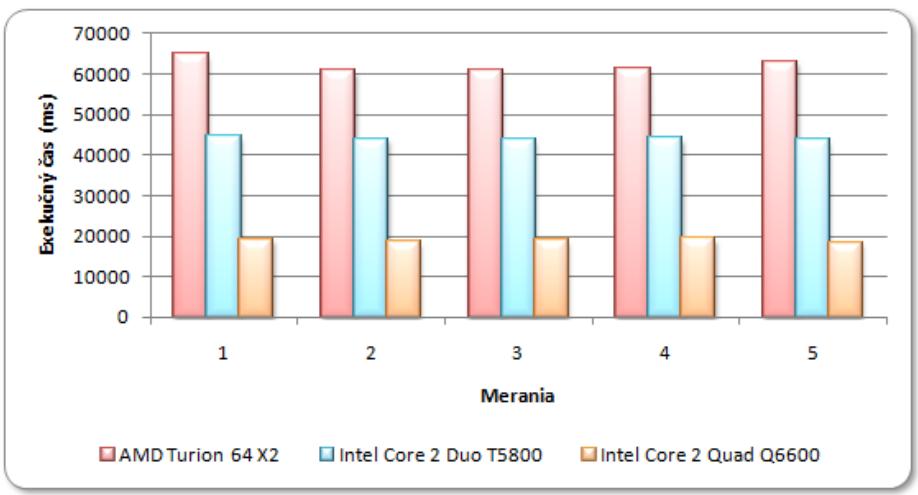
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}52}{4} \times 100 = \mathbf{88}\ \%$$

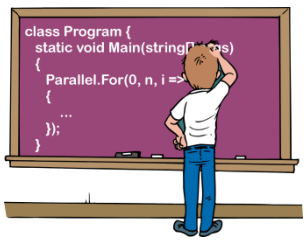


Obr. 13: Výkonnosť sekvenčného programu na testovacích počítačoch



Obr. 14: Výkonnosť paralelného programu na testovacích počítačoch

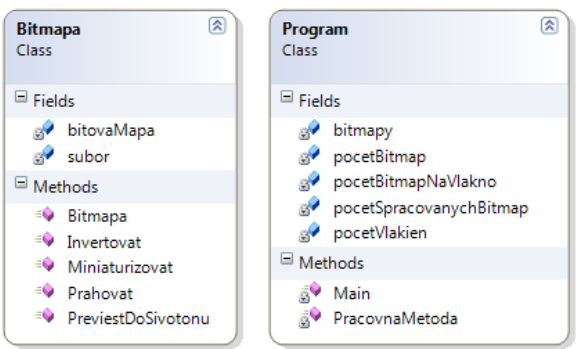
6 Praktická ukážka č. 5: Paralelné grafické transformácie bitových máp (paralelizmus s nízkou úrovňou abstrakcie)



Cieľ praktickej ukážky:
Paralelizácia grafických transformácií bitových máp.
Vedomostná náročnosť:  .
Časová náročnosť: 50 minút.
Softvérové technológie: C# 4.0 a TPL.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie:  .

Charakteristika praktickej ukážky: Táto praktická ukážka demonštruje, ako možno uskutočniť grafické transformácie kolekcie bitových máp prostredníctvom implementácie explicitného dátového paralelizmu s nízkou úrovňou abstrakcie. Miesto toho, aby sme využili programový aparát knižnice Task Parallel Library (TPL), priamo vytvárame a manipulujeme s pracovnými vláknami.

Vizualizácia tried paralelného programu:



Obr. 15: Diagram tried paralelného programu

Zdrojový kód paralelného programu:

```
using System;
using System.Collections.Generic;
```

```
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    // Deklarácia triedy Bitmapa, ktorá zapuzdruje funkcionalitu na realizáciu
    // vybraných grafických transformácií s bitovými mapami, je rovnaká
    // ako v predchádzajúcej praktickej ukážke.
    class Program
    {
        static Bitmapa[] bitmapy;
        static int pocetBitmap;
        static int pocetBitmapNaVlakno;
        static int pocetVlakien;
        static int pocetSpracovanychBitmap;
        static void Main(string[] args)
        {
            // Zistenie počtu exekučných jadier viacjadrového procesora.
            int pocetJadierCPU = Environment.ProcessorCount;
            pocetVlakien = pocetJadierCPU;
            DirectoryInfo priecinok = new DirectoryInfo(@"c:\Obrazky");
            FileInfo[] subory = priecinok.GetFiles("*.jpg");
            pocetBitmap = subory.Length;
            // Vytvorenie kolekcie bitových máp.
            bitmapy = new Bitmapa[pocetBitmap];
            Thread[] pracovneVlakna;
            // Naplnenie kolekcie bitových máp.
            for (int i = 0; i < pocetBitmap; i++)
            {
                bitmapy[i] = new Bitmapa(subory[i].FullName);
            }
            // Analýza optimálneho počtu bitových máp, ktoré bude spracúvať
            // každé pracovné vlákno.
            if (pocetBitmap < pocetVlakien)
            {
                pocetVlakien = pocetBitmap;
            }
            else if (pocetBitmap > pocetVlakien)
            {
                if (pocetBitmap % pocetVlakien == 0)
                {
                    pocetBitmapNaVlakno = pocetBitmap / pocetVlakien;
```

```
            }
            else
            {
                pocetBitmapNaVlakno = pocetBitmap / pocetVlakien + 1;
            }
        }
        // Vytvorenie kolekcie pracovných vlákien.
        pracovneVlakna = new Thread[pocetVlakien];
        // Naplnenie kolekcie pracovných vlákien.
        for (int i = 0; i < pocetVlakien; i++)
        {
            pracovneVlakna[i] = new Thread(
                new ParameterizedThreadStart(PracovnaMetoda));
        }
        Stopwatch stopky = new Stopwatch();
        Console.WriteLine("Vytvorených pracovných vlákien: {0}.",
            pocetVlakien);
        Console.WriteLine("Počet bitmáp na spracovanie: {0}.", pocetBitmap);
        Console.WriteLine("\nPrebiehajú paralelné grafické transformácie " +
            "bitových máp.");
        stopky.Start();
        // Paralelné spracovanie grafických transformácií bitových máp.
        for (int i = 0; i < pocetVlakien; i++)
        {
            pracovneVlakna[i].Start(i);
        }
        // Čakanie na dokončenie paralelného spracovania grafických
        // transformácií bitových máp.
        for (int i = 0; i < pocetVlakien; i++)
        {
            pracovneVlakna[i].Join();
        }
        stopky.Stop();
        Console.WriteLine("\nParalelné grafické transformácie " +
            "bitových máp sú hotové [čas: {0} ms].",
            stopky.ElapsedMilliseconds);
    }
    // Definícia metódy, ktorá bude vykonávaná na pracovných vláknach.
    static void PracovnaMetoda(object cisloVlakna)
    {
        // Identifikácia pracovného vlákna, s ktorým je metóda volaná.
        int vlakno = (int)cisloVlakna;
        int rozsah_min, rozsah_max;
        // Určenie segmentu bitových máp, ktoré bude spracúvať
        // každé pracovné vlákno.
        rozsah_min = vlakno * pocetBitmapNaVlakno;
        if (vlakno == pocetVlakien - 1)
```

```
            {
                rozsah_max = pocetBitmap - pocetSpracovanychBitmap;
            }
            else
            {
                rozsah_max = rozsah_min + pocetBitmapNaVlakno;
            }
            Console.WriteLine("\nSpracúva sa vlákno {0}.", vlakno + 1);
            Console.WriteLine("Vlákno {0} spracúva {1} bitmapy.", vlakno + 1,
                rozsah_max - rozsah_min);
            // Realizácia grafických transformácií bitových máp
            for (int i = rozsah_min; i < rozsah_max; i++)
            {
                bitmapy[i].Invertovat();
                bitmapy[i].PreviestDoSivotonu();
                bitmapy[i].Prahovat(128);
                Interlocked.Add(ref pocetSpracovanychBitmap, 1);
                Console.WriteLine("\nPočet spracovaných bitmáp: {0}.",
                    pocetSpracovanychBitmap);
            }
        }
    }
}
```

Verbálny súhrn zdrojového kódu paralelného programu: V záujme implementácie techniky explicitného dátového paralelizmu s nízkou úrovňou abstrakcie sme museli vykonať tieto činnosti:

1. Zistiť počet exekučných jadier viacjadrového procesora, ktorý je nainštalovaný v počítači.
2. Určiť optimálny počet pracovných vlákien.
3. Vypočítať optimálny počet bitových máp, ktoré budú jednotlivé pracovné vlákna spracúvať.
4. Definovať pracovnú metódu, ktorej inštancie budú injektované do pracovných vlákien a následne podrobené explicitnému spracovaniu.
5. Zabezpečiť optimálnu distribúciu zaťaženia naprieč pracovnými vláknami.

Primárnym cieľom nášho snaženia je navrhnúť paralelný program tak, aby vykazoval dobrú škálovateľnosť. To je nutná podmienka, pretože chceme, aby sa program dokázal dynamicky prispôsobiť variabilne výkonnému počítačovému

systému. Optimálne je, keď paralelný program obsahuje rovnaký počet pracovných vlákien ako je počet exekučných jadier viacjadrového procesora. V tomto smere existuje len jedna výnimka: ak je počet bitových máp určených na spracovanie menší ako počet pracovných vlákien, tak počet pracovných vlákien prispôsobíme počtu bitových máp. Paralelný program dbá na rovnomerné zaťaženie pracovných vlákien. V tomto ohľade je dôležitý ukazovateľ pracovného zaťaženia, ktorý je v našom prípade kvantifikovaný počtom bitových máp, ktoré spracúva jedno pracovné vlákno. Program sa snaží o rovnomerné zaťaženie pracovných vlákien. Ak úplnú rovnomernosť nie je možné dosiahnuť, tak je aplikovaný tento mechanizmus:

1. $n - 1$ pracovných vlákien disponuje rovnomerným pracovným zaťažením.
2. 1 pracovné vlákno má menšie pracovné zaťaženie ako kolekcia $n - 1$ pracovných vlákien.

V zdrojovom kóde je definovaná pracovná metóda, ktorá bude injektovaná do každého pracovného vlákna. Pracovná metóda však musí byť naprogramovaná inteligentne, pretože je potrebné, aby dokázala spolupracovať s variabilným počtom pracovných vlákien. Rovnako je nutné, aby pracovná metóda generovala optimálne zaťaženie variabilného počtu pracovných vlákien.

Pri vytváraní pracovného vlákna odovzdávame parametrickému inštančnému konštruktoru triedy **Thread** odkaz na inštanciu delegáta **ParameterizedThreadStart**, ktorá bude zapuzdrovať odkaz na cieľovú pracovnú metódu. Pracovná metóda je v našom prípade statická a rovnako parametrická. V signatúre pracovnej metódy je definovaný jeden formálny parameter typu **object**, čím je garantovaná maximálna konformita so signatúrou delegáta **ParameterizedThreadStart**. Každé pracovné vlákno bude realizovať grafické transformácie presne špecifikovaného segmentu kolekcie bitových máp. Pracovná metóda spozná pracovné vlákno (v súvislosti s ktorým je spracúvaná) podľa číselného identifikačného kódu pracovného vlákna. Tento číselný identifikačný kód je pracovnej metóde odovzdaný pri štarte pracovného vlákna (technicky – pri aktivácii inštančnej parametrickej metódy **Start** inštancie triedy **Thread**). Pracovná

metóda každému pracovnému vláknu nariadi, ktoré bitové mapy má pracovné vlákno transformovať.

6.1 Empirické testovanie paralelného programu a kvantifikácia nárastu výkonnosti

Paralelný program sme otestovali na počítačových systémoch, a to so vstupnými kolekciami 10 a 20 bitových máp. Výsledky našich meraní uvádzame v tab. 9 a v tab. 10.

Tab. 9: Výkonnosť paralelného programu pri vstupnej kolekcii 10 bitových máp

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	60904	43498	19924
2.	60659	42159	20028
3.	60497	42066	20207
4.	60406	41952	20037
5.	60558	42457	20407
$\overline{E_{TP}}$	**60605**	**42427**	**20121**

Tab. 10: Výkonnosť paralelného programu pri vstupnej kolekcii 20 bitových máp

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	122925	84600	34575
2.	124341	87246	33903
3.	131756	85476	34071
4.	146485	84872	34090
5.	132300	85145	34156
$\overline{E_{TP}}$	**131562**	**85468**	**34159**

Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 10 bitových máp na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{124270}{60605} = \mathbf{2,05}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2 je **superlineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{2,05}{2} \times 100 = \mathbf{102,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 20 bitových máp na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{242876}{131562} = \mathbf{1,85}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1{,}85}{2} \times 100 = \mathbf{92,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 10 bitových máp na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{84254}{42427} = \mathbf{1,99}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}99}{2} \times 100 = \mathbf{99,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 20 bitových máp na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{165772}{85468} = \mathbf{1,94}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}94}{2} \times 100 = \mathbf{97}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 10 bitových máp na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{66028}{20121} = \mathbf{3{,}28}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}28}{4} \times 100 = \mathbf{82}\ \%$$

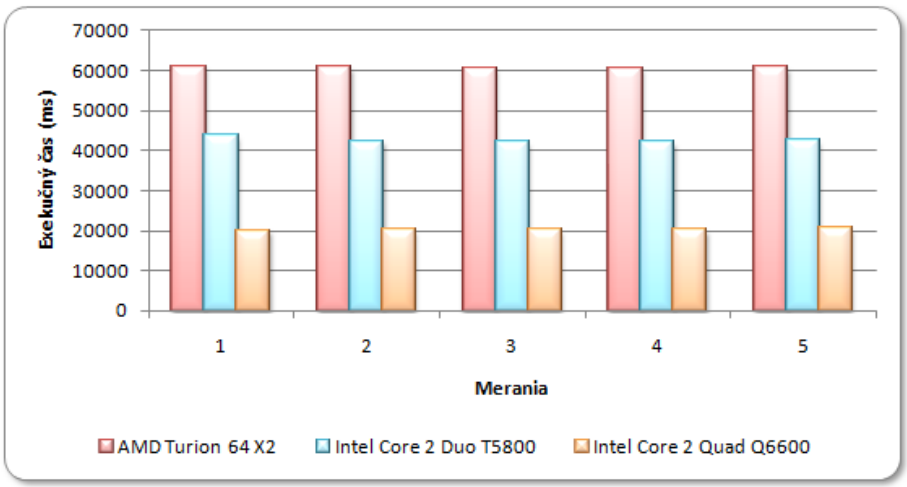
Kvantifikácia nárastu výkonnosti paralelného programu pri spracovaní kolekcie 20 bitových máp na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{132157}{34159} = \mathbf{3{,}87}$$

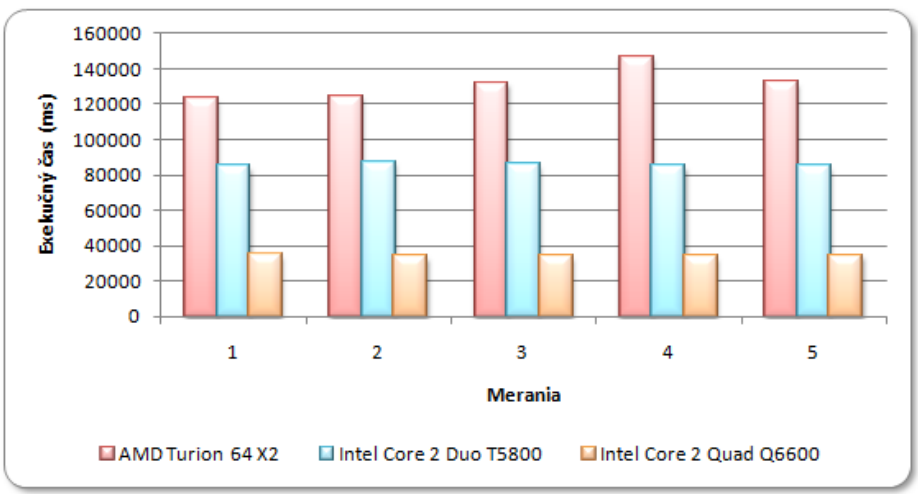
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}87}{4} \times 100 = \mathbf{96{,}75}\ \%$$



Obr. 16: Výkonnosť paralelného programu pri vstupnej kolekcii 10 bitových máp



Obr. 17: Výkonnosť paralelného programu pri vstupnej kolekcii 20 bitových máp

6.2 Diagnostika a monitorovanie výkonu paralelného programu profilovacím programom integrovaným v prostredí produktu Visual Studio 2010

Profilovací program, ktorý je integrovaný v produkte Visual Studio 2010, nám umožňuje monitorovať výkonnostné charakteristiky paralelného programu. V záujme dosiahnutia čo možno najpresnejších výsledkov odporúčame, aby bolo v operačnom systéme počas profilovania paralelného programu spustených minimum ostatných procesov.

Profilovanie paralelného programu uskutočníme takto:

1. V integrovanom vývojovom prostredí otvoríme ponuku **Analyze** a klikneme na položku **Launch Performance Wizard**.

2. V 1. kroku sprievodcu profilovacím procesom vyberieme voľbu **Concurrency**, čím vyjadríme náš záujem o diagnostiku výkonu paralelného programu. Rovnako skontrolujeme, či sú aktívne možnosti **Collect resource contention data** a **Visualize the behavior of a multithreaded application**.

3. V 2. kroku sprievodcu zvolíme aktuálny projekt paralelného programu (možnosť **One or more available projects**).

4. V 3. kroku sprievodcu sa uistíme, že je zapnutá voľba **Launch profiling after the wizard finishes**. To znamená, že profilovací proces sa automaticky spustí ihneď po ukončení sprievodcu **Performance Wizard**.

5. Nakoniec klikneme na tlačidlo **Finish**, čím spustíme profilovanie paralelného programu.

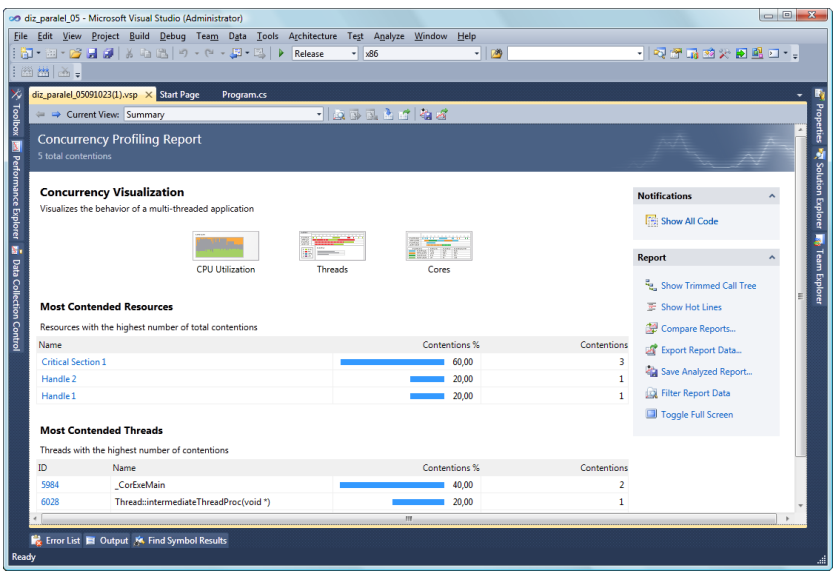
Následne dôjde k spusteniu profilovacieho programu. Profilovací program naštartuje paralelný program, ktorý má byť predmetom výkonnostného

monitorovania. Cieľom profilovacieho programu je zistiť výkonnostné charakteristiky paralelného programu. To znamená analyzovať nasledujúce atribúty paralelného programu:

1. Schopnosť paralelného programu využiť dostupnú výpočtovú kapacitu počítačového systému počas trvania jeho spracovania.
2. Zistiť počty pracovných vlákien, s ktorými paralelný program manipuluje.
3. Zistiť stavy pracovných vlákien, s ktorými paralelný program manipuluje.

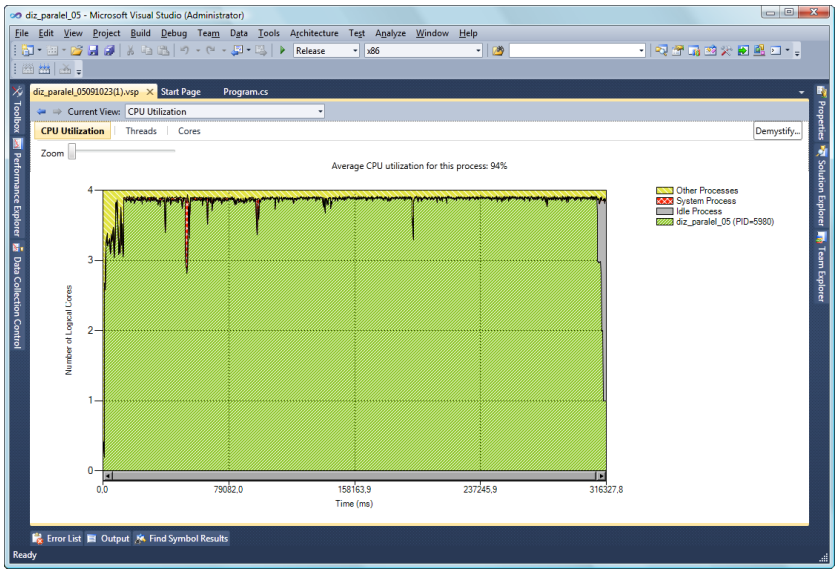
Na tomto mieste by sme radi uviedli, že profilovací program vkladá do kódu paralelného programu špeciálne fragmenty diagnostických inštrukcií. Praktickou implikáciou tohto prístupu je predĺženie exekučného času spracovania paralelného programu.

Po dokončení diagnostických akcií zobrazí profilovací program finálny výkaz **Concurrency Profiling Report**, ktorý podáva informácie o výkonnosti paralelného programu (obr. 18).



Obr. 18: Finálny výkaz, ktorý charakterizuje správanie paralelného programu

Nás zaujíma najmä schopnosť paralelného programu využiť exekučné jadrá viacjadrového procesora. Preto sa prepneme do pohľadu **CPU Utilization** (obr. 19).

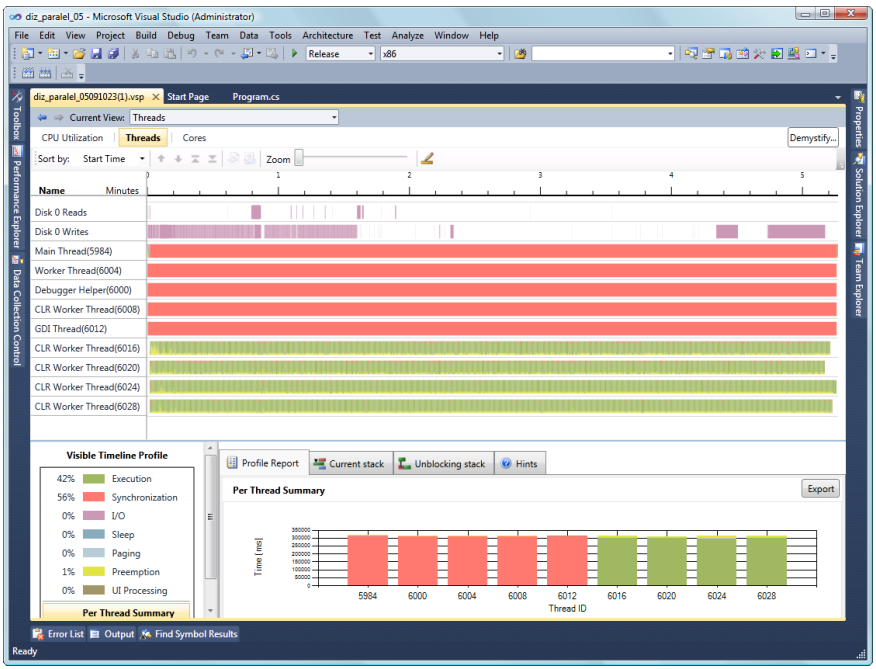


Obr. 19: Analýza miery využitia exekučných jadier viacjadrového procesora počas životného cyklu paralelného programu

Teraz opíšeme graf, ktorý je zobrazený v pohľade **CPU Utilization**. Na horizontálnej osi **Time** je nanesený exekučný čas paralelného programu (v milisekundách). Ako si môžeme všimnúť, tak profilovanie paralelného programu si vyžiadalo približne 5 minút a 16 sekúnd. Na vertikálnej osi **Number of Logical Cores** sú zaznačené exekučné jadrá viacjadrového procesora (ich číslovanie začína od nuly). Segment grafu, ktorý je zafarbený zelenou farbou, ukazuje, ako efektívne dokáže paralelný program počas svojho spracovania využívať dostupné exekučné jadrá viacjadrového procesora. Žltou farbou s jednoduchým šrafovaním sú v grafe zakreslené segmenty, ktoré predstavujú spracovanie iných procesov. Sivou farbou sú znázornené procesy, ktoré sa nachádzajú v stave nečinnosti. Červenou farbou s dvojitým šrafovaním sú vyznačené systémové procesy. Z grafu je zrejmé, že analyzovaný paralelný program je schopný počas celej svojej životnosti využívať takmer všetku výpočtovú kapacitu počítačového systému so 4-jadrovým procesorom. Po analýze zisťujeme, že

paralelný program využíval výpočtovú kapacitu počítača so 4-jadrovým procesorom v priemere na 94 % (táto informácia je viditeľná v titulnom texte **Average CPU utilization for this process**).

Keď klikneme na tlačidlo **Threads**, presunieme sa na pohľad, ktorý nám umožňuje analyzovať stavy pracovných vlákien paralelného programu počas trvania jeho životného cyklu (obr. 20).



Obr. 20: Analýza stavov pracovných vlákien

Okno pohľadu **Threads** je rozdelené na 2 časti, resp. na 2 grafy. Graf v hornej časti okna zobrazuje pracovné vlákna paralelného programu a dĺžku ich spracovania (štandardne v minútach). V grafe sa objavujú všetky vlákna, s ktorými profilovací program prišiel pri diagnostike paralelného programu do kontaktu. Pri analýze výkonnosti je preto dôležité, aby sme venovali pozornosť výhradne pracovným vláknam testovaného paralelného programu. Tieto pracovné vlákna sú jednoznačne rozpoznateľné prostredníctvom svojich číselných identifikátorov, ktoré sú nanesené

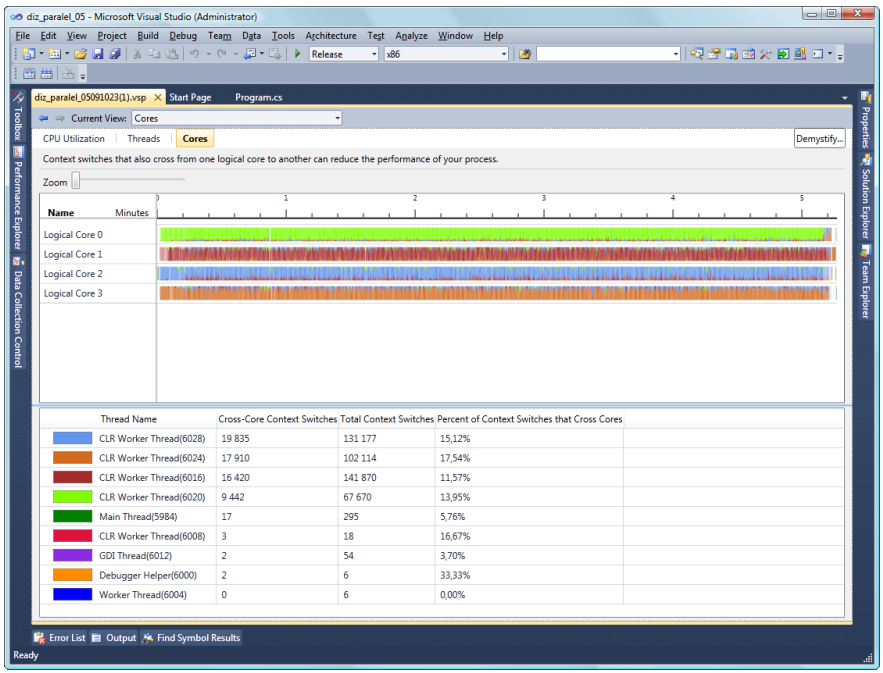
na vertikálnej osi grafu. Každému pracovnému vláknu prislúcha jeden riadok. Podľa farby riadka vieme určiť, v akom stave sa príslušné pracovné vlákno nachádza. V prípade nášho paralelného programu vidíme, že všetky pracovné vlákna sú vyplnené takmer jednoliatou zelenou farbou, čo znamená, že sa nachádzajú v stave maximálnej činnosti.

Ak nás zaujímajú stavové charakteristiky pracovných vlákien v menších časových jednotkách ako sú minúty, môžeme zobrazenie časového intervalu životnosti paralelného programu ešte viac zjemniť. To uskutočníme zvýšením magnitúdy zobrazenia pomocou bežca **Zoom**.

Graf v spodnej časti okna pohľadu **Threads** na vybratej záložke **Profile Report** a pri aktivovanej voľbe **Per Thread Summary** predstavuje kumulatívne charakteristiky stavov pracovných vlákien paralelného programu. Opäť vidíme viacero pracovných vlákien, avšak z hľadiska našej analýzy sú významné len 4 z nich (s číselnými identifikátormi 6016, 6020, 6024 a 6028). Vertikálne stĺpce grafu predstavujú kvantifikáciu času, ktorý pracovné vlákna strávili vykonávaním svojich úloh. Všetky 4 pracovné vlákna sú zaneprázdnené uskutočňovaním grafických transformácií bitových máp (táto skutočnosť je zaznamenaná použitím zelenej farby), z čoho vyplýva, že v maximálnej možnej miere využívajú dostupnú výpočtovú kapacitu počítačového systému. Segmenty vertikálnych stĺpcov, ktoré reprezentujú pracovné vlákna v grafe v spodnej časti pohľadu **Threads**, môžu byť zafarbené aj inými farbami. Napríklad, červená farba predstavuje rôzne typy synchronizačných stavov, žltá farba indikuje realizáciu preempcie a fialová farba zasa poukazuje na spracovanie vstupno-výstupných dátových operácií.

Pracovné vlákno, ktoré je aktuálne spracúvané na jednom exekučnom jadre viacjadrového procesora, môže byť transportované na iné exekučné jadro tohto procesora. Tomuto procesu vravíme migrácia pracovného vlákna. Vo všeobecnosti chceme, aby bola miera migrácie pracovných vlákien čo možno najnižšia, pretože veľký počet manipulačných operácií, ktoré sa viažu na migrovanie pracovných vlákien, má negatívny dopad na celkovú výkonnosť paralelného programu.

Posledný pohľad, ktorý predstavíme, má názov **Cores** (obr. 21).



Obr. 21: Analýza migrácie pracovných vlákien paralelného programu

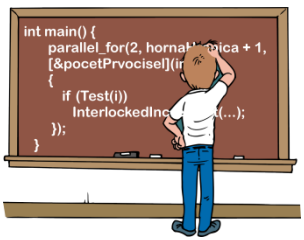
V pohľade **Cores** vidíme v hornej časti okna 1 graf, ktorý je doplnený tabuľkou umiestnenou v spodnej časti tohto okna. Najskôr budeme charakterizovať samotný graf. Na horizontálnej osi grafu je nanesený exekučný čas paralelného programu (štandardne v minútach). Na vertikálnej osi grafu sú uvedené jednotlivé exekučné jadrá viacjadrového procesora. Každé pracovné vlákno, ktoré je v grafe zakreslené, má svoju vlastnú farbu. Keďže graf skúma mieru migrácie pracovných vlákien medzi exekučnými jadrami procesora, vďaka prívetivej farebnej diferenciácii pracovných vlákien môžeme rýchlo a efektívne vypozorovať trend migrácie pracovných vlákien.

Tabuľka v spodnej časti okna pohľadu Cores podáva dátovú kvantifikáciu, ktorá diagnostikuje nasledujúce aspekty migrácie pracovných vlákien:

- Absolútna zmena kontextu pracovného vlákna pri migrácii medzi exekučnými jadrami viacjadrového procesora.
- Absolútna kumulatívna zmena kontextu pracovného vlákna.
- Relatívna zmena kontextu pracovného vlákna pri migrácii medzi exekučnými jadrami viacjadrového procesora.

Afinita pracovných vlákien s exekučnými jadrami viacjadrového procesora je v prípade skúmaného paralelného programu priaznivá, pretože stavy, kedy sú pracovné vlákna podrobené transportu medzi jadrami procesora, nie sú príliš frekventované.

7 Praktická ukážka č. 6: Vyhľadávanie prvočísel (riadený a natívny paralelizmus)



Cieľ praktickej ukážky:
Paralelizácia procesu vyhľadávania prvočísel.
Vedomostná náročnosť: .
Časová náročnosť: 25 minút.
Softvérové technológie: C# 4.0 a TPL, C++ a PPL.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie: .

Zdrojový kód sekvenčného riadeného programu v jazyku C# 4.0:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            int hornaHranica = 10000000, pocetPrvocisel = 0;
            Stopwatch stopky = new Stopwatch();
            Console.WriteLine("Prebieha sekvenčné vyhľádavanie prvočísel...");
            stopky.Start();
            // Sekvenčné vyhľadávanie prvočísel.
            for (int i = 2; i <= hornaHranica; i++)
            {
                if (Test(i)) pocetPrvocisel++;
            }
            Console.WriteLine("Sekvenčné vyhľadávanie prvočísel sa skončilo.");
            stopky.Stop();
            Console.WriteLine("V intervale <2, {0}> je {1} prvočísel.",
                hornaHranica, pocetPrvocisel);
            Console.WriteLine("Čas spracovania: {0} ms.",
                stopky.ElapsedMilliseconds);
        }
```

```
        // Metóda, ktorá uskutočňuje test prvočíselnosti.
        static bool Test(int cislo)
        {
            int odmocninaZCisla = (int)Math.Sqrt(cislo);
            for (int i = 2; i <= odmocninaZCisla; i++)
            {
                if (cislo % i == 0) return false;
            }
            return true;
        }
    }
}
```

Verbálny súhrn zdrojového kódu sekvenčného riadeného programu jazyka C# 4.0: Sekvenčná aplikácia zisťuje počet prvočísel na základe testu prvočíselnosti, ktorý realizuje nasledujúci matematický algoritmus:

1. Nech n je prirodzené vstupné číslo.
2. Skontrolujeme, či je číslo n deliteľné ktorýmkoľvek prirodzeným číslom m z intervalu $< 2, \sqrt{n} >$.
3. Ak je n deliteľné ľubovoľným m, potom n je zložené číslo, inak je n prvočíslo.

Algoritmus diagnostikuje, či je testované číslo prvočíslo, alebo zložené číslo. Ak metódu, ktorá implementuje spomenutý algoritmus, aktivujeme na všetkých číslach z požadovaného číselného intervalu, získame celkový počet prvočísel, ktoré sa v danom intervale nachádzajú.

Sekvenčná riadená aplikácia vyhľadáva prvočísla pomocou cyklu **for**, pričom v každej iterácii cyklu volá metódu **Test**. Táto parametrická metóda prevezme odovzdané číslo a preverí, či ide o prvočíslo, alebo zložené číslo. Keď je analyzované číslo prvočíslom, metóda **Test** vracia logickú pravdu, inak je návratovou hodnotou metódy logická nepravda.

Zdrojový kód sekvenčného natívneho programu v jazyku C++:

```
#include <iostream>
#include <cmath>
#include <windows.h>

using namespace std;

bool Test(int cislo);

int wmain()
{
    int hornaHranica = 10000000, pocetPrvocisel = 0;
    unsigned int cas0, casN;
    cout << "Prebieha sekvencne vyhladavanie prvocisel..." << endl;
    cas0 = GetTickCount();
    // Sekvenčné vyhľadávanie prvočísel.
    for (int i = 2; i <= hornaHranica; i++)
    {
        if (Test(i)) pocetPrvocisel++;
    }
    cout << "Sekvencne vyhladavanie prvocisel sa skoncilo." << endl;
    casN = GetTickCount();
    cout << "V intervale <2, " << hornaHranica << "> je " << pocetPrvocisel <<
        " prvocisel." << endl;
    cout << "Cas spracovania: " << casN - cas0 << " ms." << endl;
    return 0;
}
// Funkcia, ktorá uskutočňuje test prvočíselnosti.
bool Test(int cislo)
{
    int odmocninaZCisla = (int)sqrt((float)cislo);
    for (int i = 2; i <= odmocninaZCisla; i++)
    {
        if (cislo % i == 0) return false;
    }
    return true;
}
```

Verbálny súhrn zdrojového kódu sekvenčného natívneho programu jazyka C++: Zdrojový kód sekvenčného natívneho programu, ktorý sme napísali v jazyku C++, je postavený na

rovnakých princípoch ako zdrojový kód sekvenčného riadeného programu, ktorý sme vytvorili v jazyku C# 4.0. Len sme miesto riadených programových konštrukcií upotrebili ich natívne ekvivalenty.

Zdrojový kód paralelného riadeného programu v jazyku C# 4.0:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            int hornaHranica = 10000000, pocetPrvocisel = 0;
            Stopwatch stopky = new Stopwatch();
            Console.WriteLine("Prebieha paralelné vyhľádavanie prvočísel...");
            stopky.Start();
            // Paralelné vyhľadávanie prvočísel.
            Parallel.For(2, hornaHranica + 1, i =>
            {
                if (Test(i)) Interlocked.Increment(ref pocetPrvocisel);
            });
            Console.WriteLine("Paralelné vyhľadávanie prvočísel sa skončilo.");
            stopky.Stop();
            Console.WriteLine("V intervale <2, {0}> je {1} prvočísel.",
                hornaHranica, pocetPrvocisel);
            Console.WriteLine("Exekučný čas (ms): {0}.",
                stopky.ElapsedMilliseconds);
            Console.Read();
        }
        // Metóda, ktorá uskutočňuje test prvočíselnosti.
        static bool Test(int cislo)
        {
            int odmocninaZCisla = (int)Math.Sqrt(cislo);
            for (int i = 2; i <= odmocninaZCisla; i++)
            {
                if (cislo % i == 0) return false;
            }
```

```
            return true;
        }
    }
}
```

Verbálny súhrn zdrojového kódu paralelného riadeného programu jazyka C# 4.0: Región, v ktorom je aplikovaný explicitný dátový paralelizmus s vysokou úrovňou abstrakcie, je tvorený paralelným cyklom **for**. Paralelný cyklus **for** je riadený λ-príkazom. Keďže zväzky iterácií paralelného cyklu **for** budú spracúvané súbežne, je nutné predísť kolíznemu stavu, v rámci ktorého by sa viaceré pracovné vlákna snažili modifikovať hodnotu premennej **pocetPrvocisel**. Preto inkrementáciu tejto premennej realizujeme ako atomickú operáciu. Za týmto účelom voláme statickú metódu **Increment** statickej triedy **Interlocked** z menného priestoru **System.Threading**.

Zdrojový kód paralelného natívneho programu v jazyku C++:

```
#include <iostream>
#include <cmath>
#include <windows.h>
#include <ppl.h>

using namespace std;
using namespace Concurrency;

bool Test(int cislo);

int wmain()
{
    int hornaHranica = 10000000; long pocetPrvocisel = 0;
    unsigned int cas0, casN;
    cout << "Prebieha paralelne vyhladavanie prvocisel..." << endl;
    cas0 = GetTickCount();
    // Paralelné vyhľadávanie prvočísel uskutočňuje paralelný cyklus for.
    parallel_for(2, hornaHranica + 1, [&pocetPrvocisel](int i)
    {
        if (Test(i))
            InterlockedIncrement(&pocetPrvocisel);
    });
    cout << "Paralelne vyhladavanie prvocisel sa skoncilo." << endl;
    casN = GetTickCount();
    cout << "V intervale <2, " << hornaHranica << "> je " << pocetPrvocisel <<
```

```
        " prvocisel." << endl;
    cout << "Cas spracovania: " << casN - cas0 << " ms." << endl;
    return 0;
}
// Funkcia, ktorá uskutočňuje test prvočíselnosti.
bool Test(int cislo)
{
    int odmocninaZCisla = (int)sqrt((float)cislo);
    for (int i = 2; i <= odmocninaZCisla; i++)
    {
        if (cislo % i == 0) return false;
    }
    return true;
}
```

Verbálny súhrn zdrojového kódu paralelného natívneho programu jazyka C++: Tento paralelný program implementuje explicitný natívny dátový paralelizmus prostredníctvom paralelných syntakticko-sémantických konštrukcií knižnice Microsoft Parallel Patterns Library (PPL). Direktívou predprocesora **#include** zavádzame odkaz na hlavičkový súbor ppl.h. Direktívou prekladača **using namespace** explicitne sprístupňujeme všetky entity z menného priestoru **Concurrency**.

Jadrom zapracovanej natívnej paralelnej funkcionality je paralelný cyklus **for**. Technicky je paralelný cyklus **for** reprezentovaný preťaženou šablónovou funkciou s identifikátorom **parallel_for**, ktorá je definovaná v mennom priestore **Concurrency**. Všeobecná podoba prvej verzie preťaženej šablónovej funkcie **parallel_for** má nasledujúci syntaktický obraz:

```
// Definícia 1. verzie preťaženej šablónovej funkcie parallel_for.
template <typename _Index_type, typename _Function>
_Function parallel_for(_Index_type _First, _Index_type _Last, _Function _Func)
{
    return parallel_for(_First, _Last, _Index_type(1), _Func);
}
```

kde:

- **_First** je inkluzívna začiatočná hodnota iterácie paralelného cyklu **for**.
- **_Last** je exkluzívna konečná hodnota iterácie paralelného cyklu **for**.

- **_Func** je funkčný objekt (funktor), ktorý bude riadiť každú iteráciu paralelného cyklu **for**.

Návratovou hodnotou uvedenej verzie preťaženej šablónovej funkcie **parallel_for** je kópia funkčného objektu. Táto kópia je získaná vtedy, keď bol funkčný objekt aplikovaný na všetky entity, s ktorými paralelný cyklus **for** manipuloval vo svojich iteráciách.

Deklarácia triedy funkčného objektu a jej inštanciácia sú operácie, ktoré sú implicitne riadené pomocou λ-výrazu. λ-výrazy sú jednou z najvýznamnejších syntakticko-sémantických inovácií novo pripravovaného ISO štandardu jazyka C++ (štandard zatiaľ disponuje pracovným označením C++0x). Prekladač jazyka C++, ktorý je začlenený v produkte Visual C++ 2010, prácu s λ-výrazmi podporuje. λ-výraz umožňuje definovať anonymnú funkciu, ktorá je známa ako λ-funkcia. V prípade nášho paralelného programu je λ-výraz tretím argumentom, ktorý je odovzdávaný šablónovej funkcii **parallel_for**. Definícia λ-výrazu je zložená z týchto častí:

1. **[&pocetPrvocisel] je inicializačný symbol λ-výrazu.** Inicializačný symbol λ-výrazu identifikuje začiatok λ-výrazu, pričom determinuje explicitné získanie odkazu na lokálnu premennú **pocetPrvocisel**. **[]** je uvádzací symbol, ktorý vymedzuje začiatok λ-výrazu. λ-výraz môže získať prístup k ľubovoľnej automatickej lokálnej premennej, ktorá sa vyskytuje v oblasti platnosti, v ktorej je λ-výraz definovaný. λ-výraz môže k lokálnym premenným pristupovať dvomi spôsobmi: hodnotou (alokujú sa kópie lokálnych premenných) a odkazom (k lokálnym premenným sa pristupuje priamo). Ak sa v inicializačnom symbole λ-výrazu objavuje pred identifikátorom lokálnej premennej unárny referenčný operátor (&), tak ide o explicitné získanie odkazu na príslušnú premennú.

2. **(int i) je 1-prvkový zoznam formálnych parametrov λ-výrazu.** λ-výraz môže spolupracovať so zoznamom formálnych parametrov, ktorý má podobné charakteristiky ako zoznam formálnych parametrov štandardných

funkcií či členských funkcií jazyka C++. Každý správne definovaný formálny parameter λ-výrazu musí disponovať svojím identifikátorom a dátovým typom. Pre definovanie zoznamu formálnych parametrov v λ-výraze platia tieto obmedzenia:

- v zozname sa nesmú vyskytovať implicitné formálne parametre,
- v zozname sa nesmú objavovať nepomenované formálne parametre,
- počet formálnych parametrov musí byť konečný.

Navzdory tomu, že v našom λ-výraze je situovaný jeden formálny parameter celočíselného dátového typu, zoznam formálnych parametrov je len fakultatívnou súčasťou λ-výrazov.

3. **{ } je telo λ-výrazu.** Telo λ-výrazu formuje blok programových príkazov, ktoré pôsobia ako príkazy anonymne definovanej λ-funkcie. V prípade nášho paralelného programu sa v tele λ-výrazu nachádza jeden rozhodovací príkaz **if**, ktorý testuje, či analyzované číslo (uskladnené vo formálnom parametri **i** λ-výrazu) je prvočíslo, alebo nie. Ak je test prvočíselnosti pozitívny, dochádza k atomickej inkrementácii explicitne odkazovanej lokálnej premennej **pocetPrvocisel**. Atomická inkrementácia je realizovaná volaním funkcie **InterlockedIncrement**.

7.1 Empirické testovanie riadeného sekvenčného programu, natívneho sekvenčného programu, riadeného paralelného programu a natívneho paralelného programu s kvantifikáciou nárastu výkonnosti

Riadený sekvenčný program, natívny sekvenčný program, riadený paralelný program a natívny paralelný program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 11 – 14.

Tab. 11: Výkonnosť riadeného sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	38375	19714	16465
2.	38217	19755	16437
3.	38168	19732	16445
4.	38193	19731	16442
5.	38143	19717	16435
$\overline{E_{TS}}$	**38220**	**19730**	**16445**

Tab. 12: Výkonnosť natívneho sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	38158	19688	16380
2.	38127	19688	16427
3.	38127	19672	16442
4.	38158	16671	16427
5.	38158	19687	16411
$\overline{E_{TS}}$	**38146**	**19082**	**16418**

Tab. 13: Výkonnosť riadeného paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	19302	9950	4220
2.	19332	9937	4196
3.	19286	9964	4207
4.	19322	9924	4212
5.	19294	10004	4218
$\overline{E_{TP}}$	**19308**	**9956**	**4211**

Tab. 14: Výkonnosť natívneho paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	19516	9907	4197
2.	19359	9984	4196
3.	19531	10031	4228
4.	19267	9969	4212
5.	19313	9954	4196
$\overline{E_{TP}}$	**19398**	**9969**	**4206**

Kvantifikácia nárastu výkonnosti riadeného paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{38220}{19308} = \mathbf{1,98}$$

Nárast výkonnosti riadeného paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion X2 takáto:

$$e = \frac{1{,}98}{2} \times 100 = \mathbf{99}\ \%$$

Kvantifikácia nárastu výkonnosti natívneho paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{38146}{19398} = \mathbf{1,97}$$

Nárast výkonnosti natívneho paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion X2 takáto:

$$e = \frac{1{,}97}{2} \times 100 = \mathbf{98{,}5}\ \%$$

Kvantifikácia nárastu výkonnosti riadeného paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{19730}{9956} = \mathbf{1,98}$$

Nárast výkonnosti riadeného paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}98}{2} \times 100 = \mathbf{99}\ \%$$

Kvantifikácia nárastu výkonnosti natívneho paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{19082}{9969} = \mathbf{1{,}91}$$

Nárast výkonnosti natívneho paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}91}{2} \times 100 = \mathbf{95{,}5}\ \%$$

Kvantifikácia nárastu výkonnosti riadeného paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{16445}{4211} = \mathbf{3{,}91}$$

Nárast výkonnosti riadeného paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}91}{4} \times 100 = \mathbf{97{,}75}\ \%$$

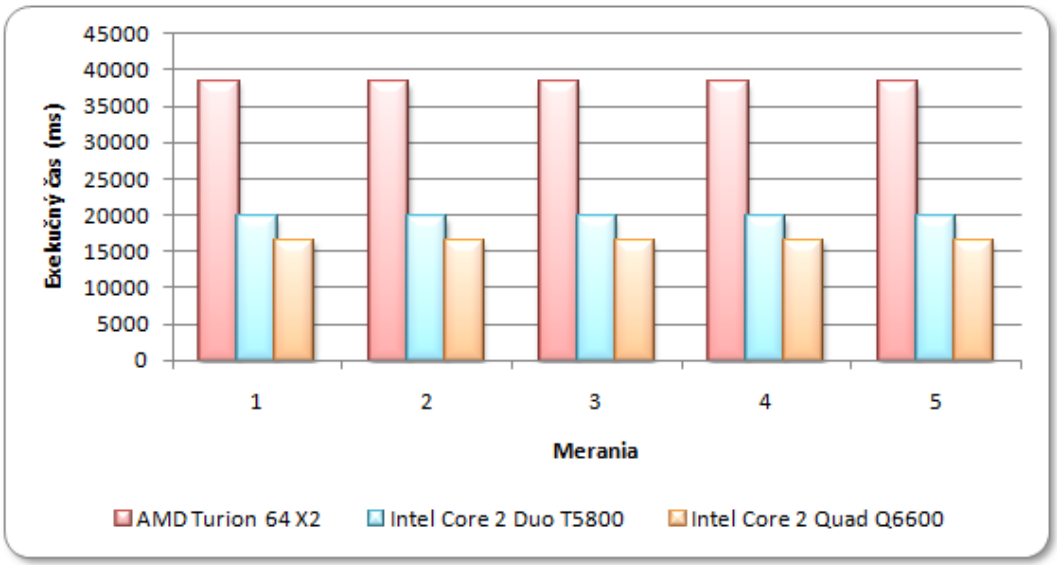
Kvantifikácia nárastu výkonnosti natívneho paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{16418}{4206} = \mathbf{3{,}90}$$

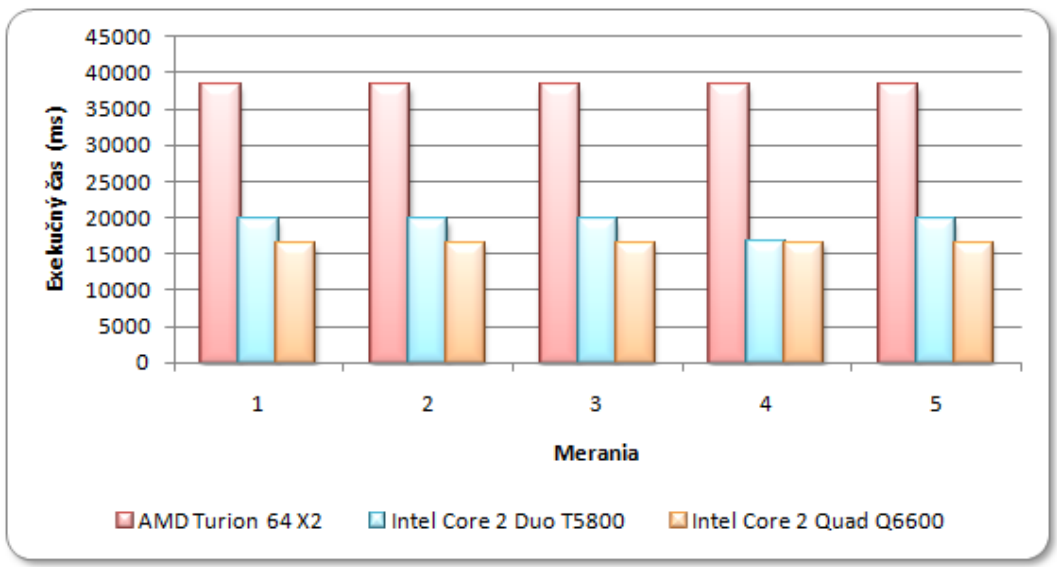
Nárast výkonnosti riadeného paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

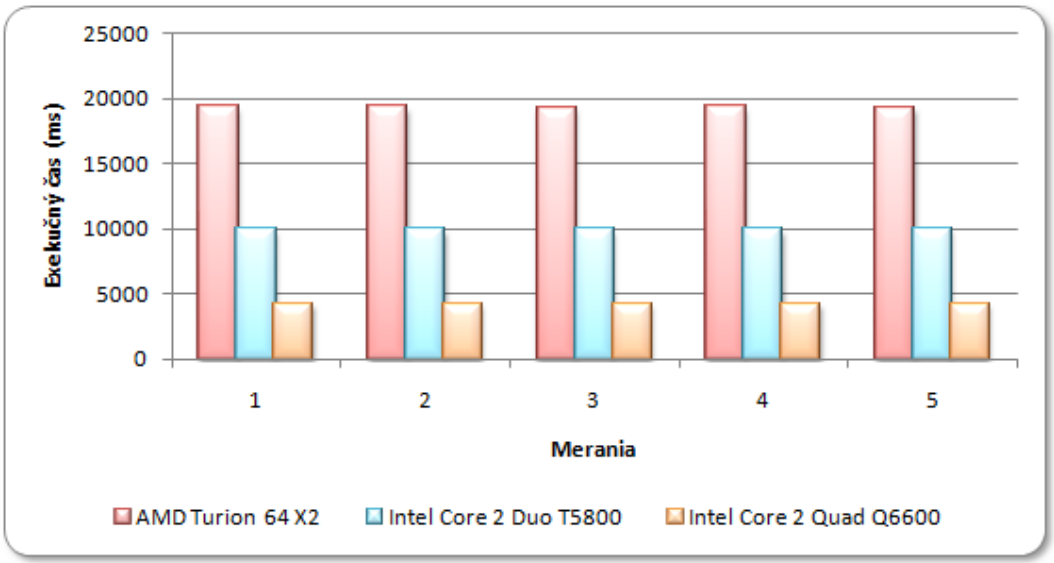
$$e = \frac{3{,}90}{4} \times 100 = \mathbf{97{,}5}\ \%$$



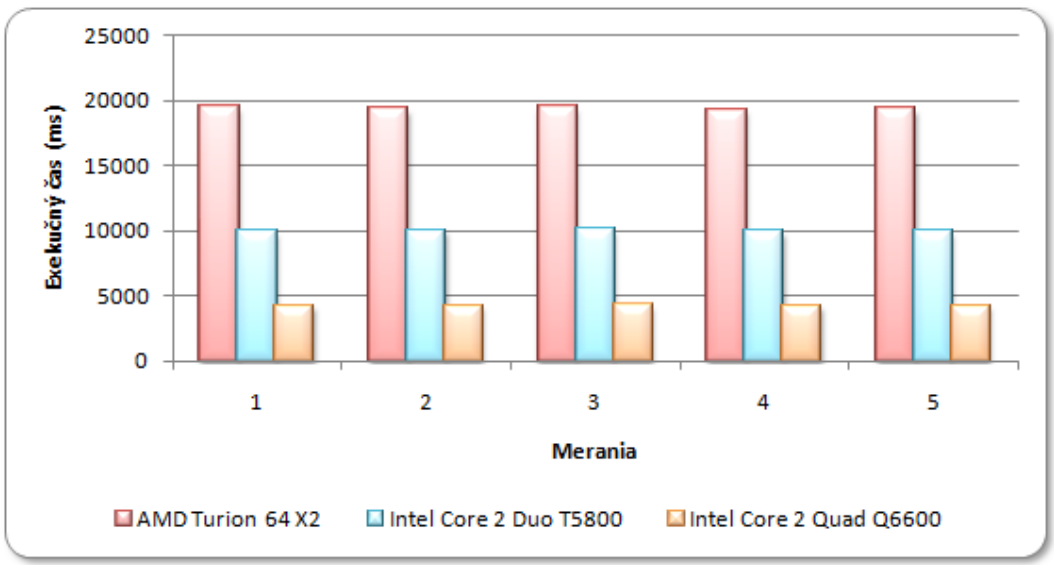
Obr. 22: Výkonnosť riadeného sekvenčného programu na testovacích počítačoch



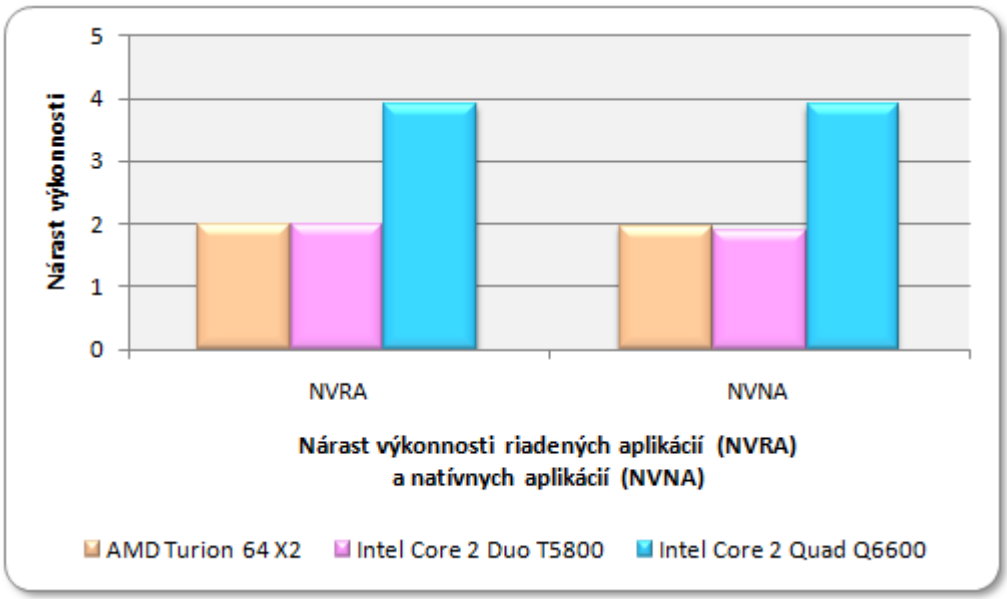
Obr. 23: Výkonnosť natívneho sekvenčného programu na testovacích počítačoch



Obr. 24: Výkonnosť riadeného paralelného programu na testovacích počítačoch

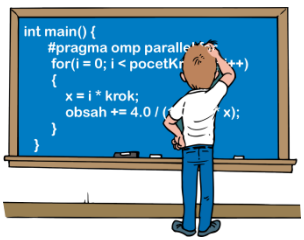


Obr. 25: Výkonnosť natívneho paralelného programu na testovacích počítačoch



Obr. 26: Nárast výkonnosti riadených a natívnych paralelných aplikácií

8 Praktická ukážka č. 7: Numerická integrácia – natívny paralelizmus pomocou rozhrania OpenMP



Cieľ praktickej ukážky:
Paralelizácia numerickej integrácie pri výpočte aproximovanej hodnoty určitého integrálu $\int_0^1 \frac{4}{1+x^2} dx$, pričom platí, že $\int_0^1 \frac{4}{1+x^2} dx \cong \pi$.
Vedomostná náročnosť: .
Časová náročnosť: 30 minút.

Softvérové technológie: C++ a OpenMP.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie: .

Zdrojový kód sekvenčného natívneho programu v jazyku C++:

```
#include <iostream>
#include <windows.h>

using namespace std;

int wmain()
{
    double krok, x, obsah = 0.0;
    long int i, pocetKrokov = 1000000000;
    unsigned int cas0, casN;
    krok = 1.0 / pocetKrokov;
    for(int pocetRelacii = 1; pocetRelacii <= 5; pocetRelacii++)
    {
        x = obsah = 0.0;
        cas0 = GetTickCount();
        cout << "Vypocet c. " << pocetRelacii << " sa zacal." << endl;
        // Sekvenčný algoritmus numerickej integrácie.
        for(i = 0; i < pocetKrokov; i++)
        {
            x = i * krok;
            obsah += 4.0 / (1.0 + x * x);
        }
        casN = GetTickCount();
        cout << "Vypocet c. " << pocetRelacii << " sa skoncil." << endl;
        cout.precision(20);
```

```
        cout << "Vystup: " << obsah * krok << "." << endl;
        cout << "Exekucny cas: " << casN - cas0 << " ms." << endl << endl;
    }
    cout << "Sekvencna numericka integracia je hotova." << endl;
    return 0;
}
```

Verbálny súhrn zdrojového kódu sekvenčného natívneho programu: Sekvenčná verzia pracuje s 1 miliardou parciálnych intervalov. Program meria exekučný čas (v ms), ktorý numerická analýza alokuje, a vzápätí ho zobrazuje používateľovi. Na výstup je odosielaná aj vypočítaná hodnota Ludolfovho čísla (π). Výpočet numerickej integrácie uskutočňujeme dovedna v 5 reláciách.

Zdrojový kód paralelného natívneho programu v jazyku C++:

```
#include <iostream>
// Import hlavičkového súboru rozhrania OpenMP.
#include <omp.h>
#include <windows.h>

using namespace std;
int wmain()
{
    double krok, x, obsah = 0.0;
    long int i, pocetKrokov = 1000000000;
    unsigned int cas0, casN;
    krok = 1.0 / pocetKrokov;
    for(int pocetRelacii = 1; pocetRelacii <= 5; pocetRelacii++)
    {
        x = obsah = 0.0;
        cas0 = GetTickCount();
        cout << "Vypocet c. " << pocetRelacii << " sa zacal." << endl;
        // Paralelný algoritmus numerickej integrácie.
        #pragma omp parallel for reduction(+: obsah) private(x)
        for(i = 0; i < pocetKrokov; i++)
        {
            x = i * krok;
            obsah += 4.0 / (1.0 + x * x);
        }
        casN = GetTickCount();
        cout << "Vypocet c. " << pocetRelacii << " sa skoncil." << endl;
        cout.precision(20);
```

```
        cout << "Vystup: " << obsah * krok << "." << endl;
        cout << "Exekucny cas: " << casN - cas0 << " ms." << endl << endl;
    }
    cout << "Paralelna numericka integracia je hotova." << endl;
    return 0;
}
```

Verbálny súhrn zdrojového kódu paralelného natívneho programu: Zmeny, ktoré sme do zdrojového kódu zapracovali, sa týkajú dvoch hlavných oblastí:

1. Direktívou predprocesora **#include** zavádzame odkaz na hlavičkový súbor omp.h rozhrania OpenMP.

2. Direktívou kompilátora **#pragma omp parallel for** identifikujeme paralelnú sekciu, ktorá je tvorená iteratívnym príkazom **for**. Ako si môžeme všimnúť, v direktíve aplikujeme klauzuly **reduction** a **private**.

Klauzula **reduction** sa viaže na premennú **obsah**, v ktorej uchovávame postupne vypočítané obsahy obdĺžnikov, ktorými aproximujeme hodnotu určitého integrálu. Keďže zväzky iterácií paralelného cyklu **for** budú spracúvané súbežne, musíme sa vyhnúť potenciálnym pretekom pracovných vlákien, ktoré by viedli k nedeterministickým výsledkom. Povedané inak, je našou povinnosťou zabezpečiť bezpečný priebeh inkrementačných operácií, ktoré sú aplikované na premennú **obsah**. Vzhľadom na to, že táto premenná je modifikovaná naprieč rôznymi zväzkami paralelného cyklu **for**, musíme na ňu aplikovať tzv. redukciu. V rámci redukcie sú v paralelne vykonávaných zväzkoch cyklu vytvorené dočasné lokálne kópie požadovanej premennej, ktoré sú korektne inicializované a modifikované. Po skončení paralelnej exekúcie sú hodnoty všetkých dočasných lokálnych kópií premennej redukované do pôvodnej premennej. Redukčná klauzula používa redukčný operátor **+** s implicitnou nulovou inicializačnou hodnotou dočasných lokálnych kópií premennej.

Klauzulou **private** explicitne nariaďujeme, že špecifikovaná premenná je súkromná. To znamená, že exekučné prostredie rozhrania OpenMP vytvorí lokálne kópie požadovanej premennej pre každé pracovné vlákno.

Pre úspešné spustenie paralelnej verzie algoritmu numerickej integrácie urobíme v integrovanom vývojovom prostredí produktu Visual Studio 2010 toto:

- Ak nie je zobrazené, zviditeľníme podokno **Solution Explorer**.
- Na zdrojový súbor nášho programu jazyka C++, ktorý je umiestnený v priečinku **Source Files**, klikneme pravým tlačidlom myši a z miestnej ponuky vyberieme príkaz **Properties**.
- Po zobrazení dialógového okna **Property Pages** sa zameriame na stromovú štruktúru volieb, ktorá je situovaná na ľavej strane okna.
- V sekcii **Configuration Properties** rozvinieme uzol **C/C++** a klikneme na položku **Language**.
- V zozname napravo vyhľadáme poslednú položku **OpenMP Support** a nastavíme ju na hodnotu **Yes (/openmp)**.
- Dialógové okno **Property Pages** zatvoríme aktiváciou tlačidla **OK**.
- Preložíme zdrojový súbor a zostavíme spustiteľný súbor paralelného programu (**Build** → **Build Solution**).

8.1 Empirické testovanie sekvenčného a paralelného natívneho programu a kvantifikácia nárastu výkonnosti

Sekvenčný i paralelný natívny program sme otestovali na počítačových systémoch. Výsledky našich meraní uvádzame v tab. 15 a v tab. 16.

Tab. 15: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	8611	15765	13151
2.	8533	15750	13151
3.	8564	15750	13167
4.	8565	15766	13166
5.	8533	15766	13167
$\overline{E_{TS}}$	**8562**	**15760**	**13161**

Tab. 16: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	4305	7922	3338
2.	4290	7890	3339
3.	4305	7875	3338
4.	4306	7829	3261
5.	4274	7875	3260
$\overline{E_{TP}}$	**4296**	**7879**	**3308**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{8562}{4296} = \mathbf{1,99}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1{,}99}{2} \times 100 = \mathbf{99{,}5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{15760}{7879} = \mathbf{2{,}00}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **lineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{2{,}00}{2} \times 100 = \mathbf{100}\ \%$$

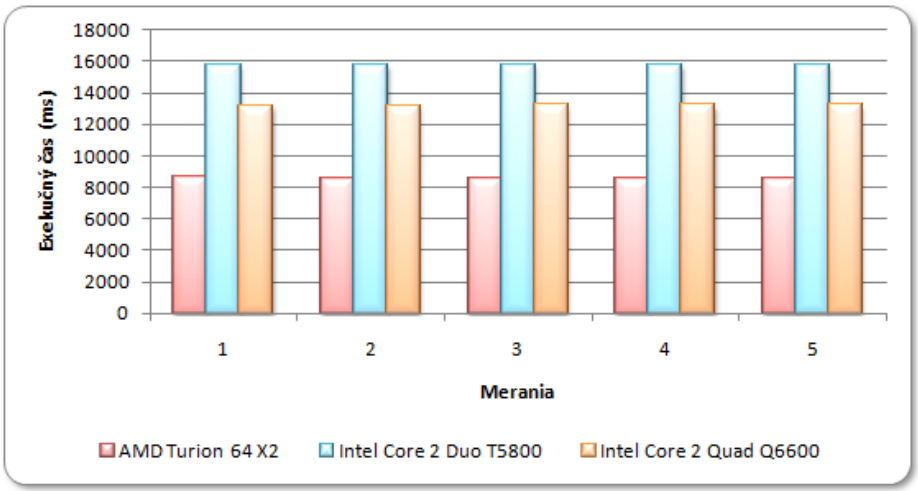
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{13161}{3308} = \mathbf{3{,}98}$$

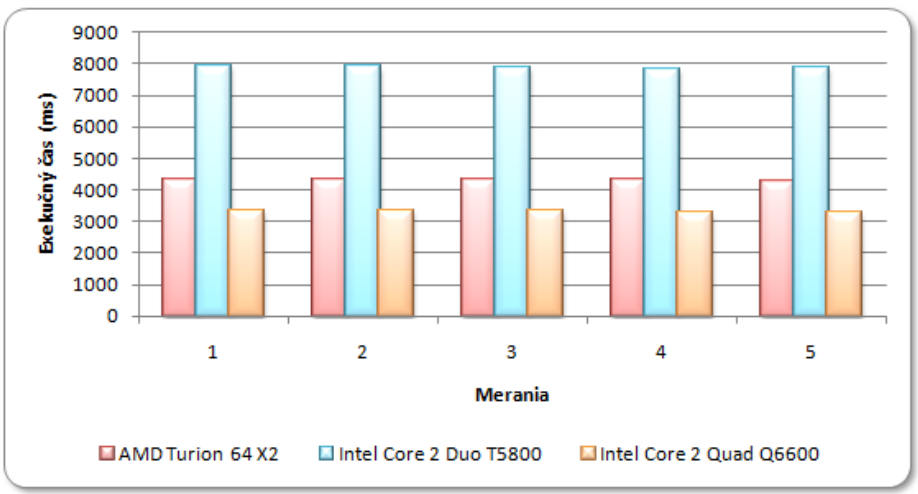
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}98}{4} \times 100 = \mathbf{99{,}5}\ \%$$



Obr. 27: Výkonnosť sekvenčného natívneho programu na testovacích počítačoch



Obr. 28: Výkonnosť paralelného natívneho programu na testovacích počítačoch

8.2 Diagnostika a monitorovanie výkonu paralelného natívneho programu profilovacím programom Intel Parallel Amplifier integrovaným v prostredí produktu Intel Parallel Studio

Keďže produkt Intel Parallel Studio nebol v čase tvorby tohto diela kompatibilný s vývojovým prostredím Visual Studio 2010, použili sme na monitorovanie výkonu paralelného natívneho programu jazyka C++ vývojové prostredie Visual Studio 2008. Diagnostika paralelného natívneho programu je realizovaná profilovacím programom s názvom Intel Parallel Amplifier. Proces profilovania paralelného natívneho programu sa odohráva v týchto krokoch:

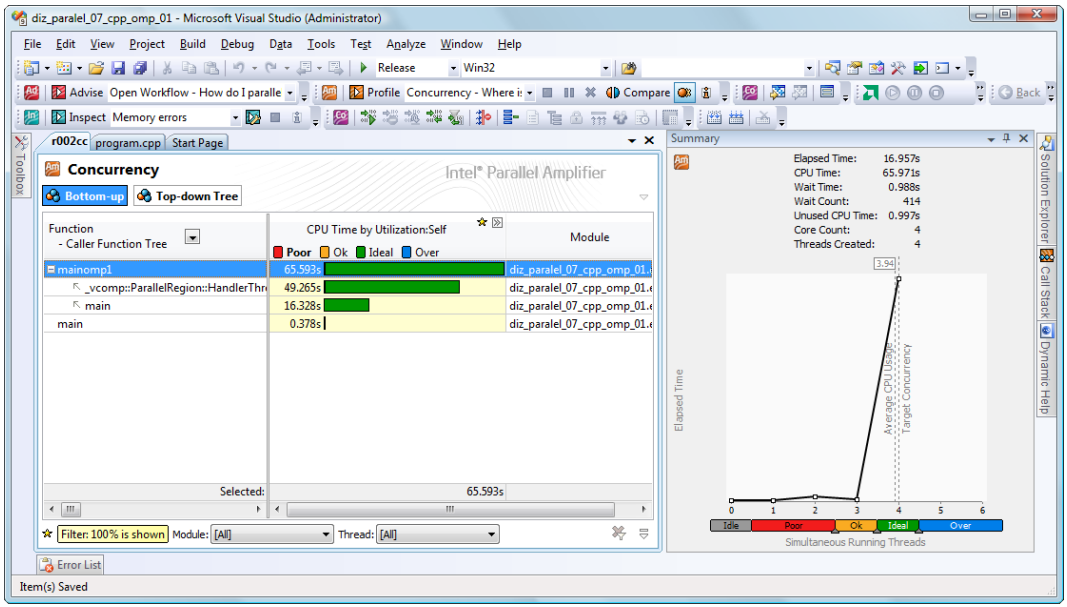
1. Do vývojového prostredia Visual Studio 2008 načítame riešenie s projektom paralelného natívneho programu.

2. Uskutočníme ostré zostavenie projektu (v zostavovacom režime **Release**).

3. Na paneli nástrojov **Intel Parallel Amplifier** zvolíme profil **Concurrency – Where is my concurrency poor?** a klikneme na tlačidlo **Profile**.

4. Profilovací program začne analýzu výkonnosti paralelného natívneho programu. V závislosti od zložitosti paralelného programu môže profilovanie trvať niekoľko desiatok sekúnd až minút.

5. Keď je analýza hotová, profilovací program zobrazí diagnostické výsledky, ku ktorým sa dopracoval (obr. 29).



Obr. 29: Výsledky analýzy výkonnosti paralelného natívneho programu profilovacím programom Intel Parallel Amplifier

Sumárne štatistiky sú prehľadne usporiadané v podokne **Summary**. Toto podokno sa skladá z 2 častí: textovej časti (nachádza sa hore) a grafickej časti (je umiestnená dole). V textovej časti sú zoskupené tieto opisné charakteristiky výkonnosti paralelného natívneho programu:

- **Elapsed Time** – celkový čas spracovania paralelného natívneho programu. Celkový čas sa vypočíta takto:

$$T = T_K - T_Z$$

kde:

  - T je celkový čas spracovania paralelného natívneho programu.
  - T_K je čas, kedy paralelný natívny program skončil svoje spracovanie.
  - T_Z je čas, kedy paralelný natívny program začal svoje spracovanie.

- **CPU Time** – celkový čas, ktorý viacjadrový procesor počítača alokoval na spracovanie paralelného natívneho programu. V našej analýze vždy predpokladáme, že paralelný natívny program bol spustený na počítačovom systéme s viacjadrovým procesorom. Za týchto okolností má dĺžka tohto časového ukazovateľa kumulatívny charakter. To znamená, že časy, ktoré strávia jednotlivé exekučné jadrá viacjadrového procesora vykonávaním paralelných výpočtov, sa spočítavajú. Ak napríklad 1. jadro vykonávalo paralelné výpočty 4 sekundy, 2. jadro 3 sekundy, 3. jadro 5 sekúnd a 4. jadro 4 sekundy, tak ukazovateľ celkového procesorového času bude rovný 16 sekundám. Dodajme, že ukazovateľ **CPU Time** bude spravidla nadobúdať väčšie hodnoty ako ukazovateľ **Elapsed Time**.

- **Wait Time** – celkový čas, počas ktorého programové vlákna paralelného natívneho programu čakajú na istú udalosť. Touto udalosťou môže byť napríklad čakanie na získanie zámku ku kritickej sekcii či čakanie na dokončenie vstupno-výstupných dátových operácií.

- **Wait Count** – celkový počet systémových volaní, ktoré generovali čakanie programových vlákien paralelného natívneho programu.

- **Unused CPU Time** – celkový čas, ktorý determinuje, ako dlho sú nevyužité výpočtové kapacity viacjadrového procesora. Predpokladajme, že na počítači so 6-jadrovým procesorom je spustený 2-vláknový paralelný program. Celková dĺžka spracovania paralelného programu nech je 10 sekúnd. Keďže každé vlákno paralelného programu bude mapované na práve jedno exekučné jadro 6-jadrového procesora počas 10 sekúnd, znamená to, že využité budú iba 2 zo 6 dostupných exekučných jadier procesora. Povedané inak, 4 jadrá procesora sú v priebehu 10-sekundového časového intervalu nečinné, a teda nevyužité. Ukazovateľ **Unused CPU Time** bude v tomto prípade generovať hodnotu 40 s.

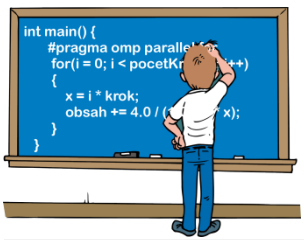
- **Core Count** – celkový počet logických procesorov počítača. Ak je paralelný natívny program spustený na počítači s viacjadrovým procesorom bez

technológie Intel Hyper-Threading (HT), tak sa počet logických procesorov rovná počtu exekučných jadier viacjadrového procesora. Naopak, keď je paralelný natívny program spracovaný na stroji s viacjadrovým procesorom s podporou HT technológie, potom sa počet logických procesorov rovná dvojnásobku počtu exekučných jadier viacjadrového procesora.

- **Threads Created** – celkový počet programových vlákien paralelného natívneho programu.

Dvojrozmerný spojnicový graf uložený v grafickej časti podokna **Summary** ponúka intuitívnu vizualizáciu výkonu paralelného natívneho programu. Na horizontálnej osi grafu sledujeme celkový počet súbežne spracúvaných programových vlákien. Na vertikálnej osi grafu je zaznačený celkový čas spracovania paralelného natívneho programu. Profilovací program klasifikuje výkonnosť paralelného programu pomocou farebných segmentov. Naším cieľom je, aby paralelný program trávil čo možno najviac času v zelenom segmente, teda v časovom intervale, keď sú všetky pracovné vlákna programu aktívne spracúvané na dostupných exekučných jadrách viacjadrového procesora. Profilovací program vypočíta aj ukazovateľ priemerného využitia výpočtovej kapacity stroja paralelným natívnym programom. Z uskutočnenej analýzy vyplýva, že náš paralelný natívny program počas 98,5 % celkového času svojho spracovania alokoval všetky exekučné jadrá 4-jadrového procesora Intel Core 2 Quad Q6600. To predstavuje sublineárny nárast výkonnosti s koeficientom 3,94.

9 Praktická ukážka č. 8: Implicitný natívny paralelizmus – automatická paralelizácia algoritmu numerickej integrácie pomocou prekladača Intel C++ Compiler 11.1



Cieľ praktickej ukážky:
Preskúmať schopnosti automatickej paralelizácie prekladača Intel C++ Compiler 11.1 pri implementácii algoritmu numerickej integrácie za účelom výpočtu aproximovanej hodnoty určitého integrálu $\int_0^1 \frac{4}{1+x^2} dx$, pričom platí, že $\int_0^1 \frac{4}{1+x^2} dx \cong \pi$.
Vedomostná náročnosť: .
Časová náročnosť: 15 minút.
Softvérové technológie: C++, automatický paralelizér prekladača Intel C++ Compiler 11.1.
Druh paralelizmu: Explicitný dátový paralelizmus.
Úroveň abstrakcie: .

Implicitný paralelizmus predstavuje automatickú paralelizáciu zdrojového kódu sekvenčného programu. Automatickú paralelizáciu má na starosti prekladač, čo znamená, že celý proces paralelizácie sekvenčného programu je v jeho kompetencii a nie je teda vyžadovaná žiadna intervencia zo strany vývojára. Proces automatickej paralelizácie sekvenčného kódu sa skladá z nasledujúcich etáp:

1. Analýza zdrojového kódu sekvenčného programu za účelom detekcie potenciálne paralelizovateľných regiónov kódu.

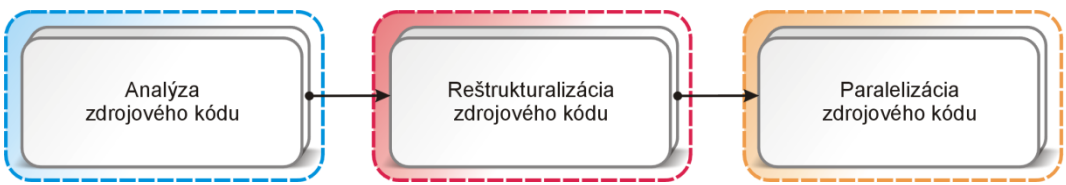
2. Reštrukturalizácia zdrojového kódu sekvenčného programu pre uľahčenie paralelizácie vhodných regiónov kódu.

3. Paralelizácia zdrojového kódu, teda transformácia vhodných regiónov sekvenčného kódu na ekvivalentný paralelný kód.

Automatická paralelizácia sa spravidla koncentruje na dátový paralelizmus. Prekladač najskôr vyhľadá všetky programové cykly, a potom vykoná analýzu, podľa ktorej zistí, ktoré cykly sú vhodnými kandidátmi na paralelizáciu. Prekladač pritom

berie do úvahy statickú a dynamickú analýzu dátových závislostí medzi jednotlivými iteráciami cyklov. Samozrejme, najpriaznivejšia situácia nastáva vtedy, keď medzi iteráciami cyklu neexistujú žiadne priame dátové závislosti. Za týchto okolností môže prekladač emitovať inštrukcie, ktoré zabezpečia rozdelenie všetkých iterácií cyklu do vyvážených zväzkov iterácií a následné paralelné spracovanie týchto zväzkov iterácií. Radi by sme prízvukovali, že prekladač uskutoční automatickú paralelizáciu len tých programových cyklov, pri ktorých bude túto transformáciu považovať za bezpečnú a efektívnu. To znamená, že pre úspešnú paralelizáciu cyklu musia prínosy plynúce z paralelizácie v čo možno najväčšej miere prevyšovať náklady spojené s procesom paralelizácie. Ak prekladač po vyhodnotení svojich analýz zistí, že paralelizácia cyklu by neprinášala očakávaný nárast výkonnosti (alebo by dokonca spôsobovala pokles aktuálnej výkonnosti cyklu), tak zdrojový kód ponechá v pôvodnej, a teda sekvenčnej forme.

Z technického hľadiska zabezpečuje automatickú paralelizáciu rozhranie OpenMP. Kým pri explicitnom paralelizme je to vývojár, kto identifikuje paralelné sekcie pomocou direktív rozhrania OpenMP, pri implicitnom paralelizme tieto činnosti spadajú do kompetencie prekladača s podporou automatickej paralelizácie.



Obr. 30: Proces automatickej paralelizácie zdrojového kódu sekvenčného programu

V tejto praktickej ukážke je naším cieľom overiť, aký nárast výkonnosti prinesie automatická paralelizácia sekvenčného programu, ktorý implementuje algoritmus numerickej integrácie riešiaci aproximovaný výpočet Ludolfovho čísla. Implicitný paralelizmus do sekvenčného programu zavedieme pomocou prekladača Intel C++ Compiler 11.1. Keďže v čase tvorby tohto diela sa prekladač Intel C++ Compiler 11.1 neintegroval do vývojového prostredia produktu Visual Studio 2010, použili sme pri testoch produkt Visual Studio 2008.

Zdrojový kód sekvenčného programu jazyka C++:

```
#include <iostream>
#include <windows.h>

using namespace std;

int main()
{
    double krok, x, obsah = 0.0;
    long int i, pocetKrokov = 1000000000;
    unsigned int cas0, casN;
    krok = 1.0 / pocetKrokov;
    x = obsah = 0.0;
    cas0 = GetTickCount();
    for(i = 0; i < pocetKrokov; i++)
    {
        x = i * krok;
        obsah += 4.0 / (1.0 + x * x);
    }
    casN = GetTickCount();
    cout.precision(20);
    cout << "Vystup: " << obsah * krok << "." << endl;
    cout << "Exekucny cas: " << casN - cas0 << " ms." << endl << endl;
    cout << "Sekvencna numericka integracia je hotova." << endl;
    return 0;
}
```

V ďalšom kroku aktivujeme automatickú paralelizáciu prekladača. Túto akciu vo vývojovom prostredí Visual Studio 2008 uskutočníme nasledujúcim spôsobom:

- V podokne **Solution Explorer** vyhľadáme priečinok **Source Files** a klikneme pravým tlačidlom myši na súbor so zdrojovým kódom sekvenčného programu.
- Z miestnej ponuky vyberieme položku **Properties**.
- V stromovej štruktúre **Configuration Properties** dialógového okna **Property Pages** vyberieme uzol **C/C++** a klikneme na položku **Optimization**.
- V sekcii **Intel Specific** vyhľadáme voľbu **Parallelization** a nastavíme ju na hodnotu **Enable Parallelization**.

- Zmenu potvrdíme stlačením tlačidla **OK**.
- Uskutočníme opätovné zostavenie projektu jazyka C++ (**Build** → **Build Solution**).

Prekladač Intel C++ Compiler 11.1 vykoná automatickú paralelizáciu zdrojového kódu sekvenčného programu. Vo chvíli, keď je prekladač so svojou prácou hotový, získavame paralelný program, ktorý vznikol implementáciou implicitného dátového paralelizmu do sekvenčného programu. V tab. 17 a v tab. 18 uvádzame výsledky výkonnostných meraní sekvenčného a paralelného programu.

Tab. 17: Výkonnosť sekvenčného programu

Výpočet	Exekučný čas (E_{TS}) spracovania sekvenčného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	8829	7953	6474
2.	8783	7891	6489
3.	8814	7890	6474
4.	8814	7891	6474
5.	8829	7890	6505
$\overline{E_{TS}}$	**8814**	**7903**	**6484**

Tab. 18: Výkonnosť paralelného programu

Výpočet	Exekučný čas (E_{TP}) spracovania paralelného programu na počítači s viacjadrovým procesorom (v ms)		
	AMD Turion 64 X2	Intel Core 2 Duo T5800	Intel Core 2 Quad Q6600
1.	4462	4125	1731
2.	4509	4110	1700
3.	4478	4125	1669
4.	4477	4093	1685
5.	4493	4109	1685
$\overline{E_{TP}}$	**4484**	**4113**	**1694**

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{8814}{4484} = \mathbf{1,97}$$

Nárast výkonnosti paralelného programu na počítači s procesorom AMD Turion X2 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom AMD Turion 64 X2 takáto:

$$e = \frac{1{,}97}{2} \times 100 = \mathbf{98,5}\ \%$$

Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{7903}{4113} = \mathbf{1,92}$$

Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Duo T5800 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Duo T5800 takáto:

$$e = \frac{1{,}92}{2} \times 100 = \mathbf{96}\ \%$$

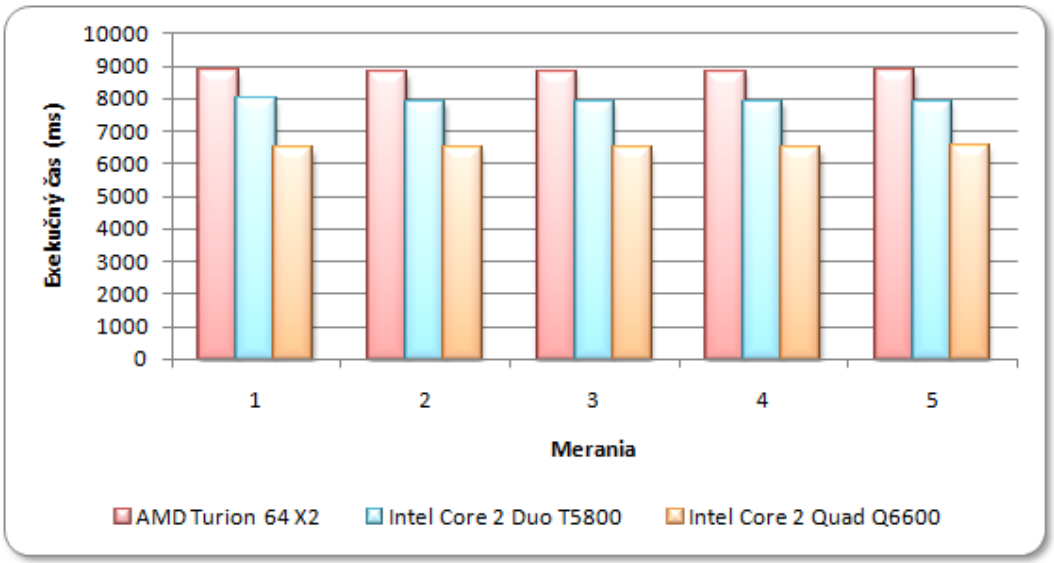
Kvantifikácia nárastu výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{6484}{1694} = \mathbf{3{,}83}$$

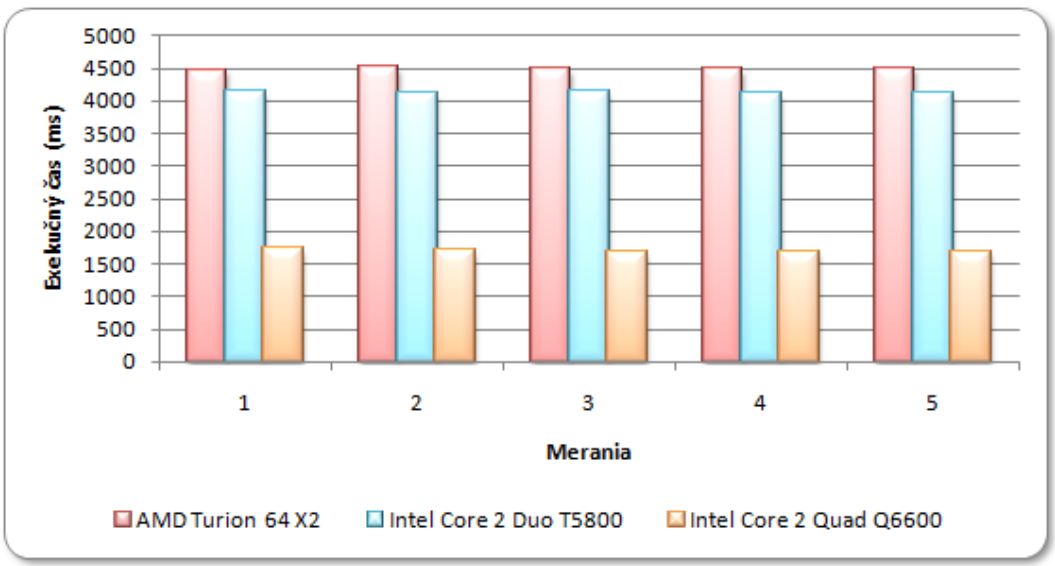
Nárast výkonnosti paralelného programu na počítači s procesorom Intel Core 2 Quad Q6600 je **sublineárny**.

Efektivita využitia výpočtových zdrojov počítača paralelným programom je na počítači s procesorom Intel Core 2 Quad Q6600 takáto:

$$e = \frac{3{,}83}{4} \times 100 = \mathbf{95{,}75}\ \%$$



Obr. 31: Výkonnosť sekvenčného programu na testovacích počítačoch



Obr. 32: Výkonnosť paralelného programu na testovacích počítačoch

Na experimentálnom zdrojovom kóde sekvenčného programu sme overili, že automatická paralelizácia generuje značný nárast výkonnosti. Preto odporúčame, aby vývojári používali pri svojej práci prekladač, ktorý je schopný automatickú paralelizáciu sekvenčného kódu uskutočniť. Samozrejme, automatická paralelizácia môže byť s výhodou kombinovaná s ďalšou optimalizačnou technikou, ktorá sa nazýva automatická vektorizácia. Zmyslom automatickej vektorizácie je využitie SIMD inštrukcií, ktoré sú schopné implementovať hardvérový paralelizmus na úrovni mikroinštrukcií.

10 Finálny sumár kvantitatívnych charakteristík praktických ukážok

V tab. 19 podávame sumárne informácie, ktoré sme zaznamenali pri empirických testoch paralelných programov vo vyvinutých praktických ukážkach.

Tab. 19: Sumárne kvantitatívne charakteristiky praktických ukážok

Praktická ukážka	N_V			e (%)		
Matematické operácie s 3D vektormi	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,61	1,77	3,34	**80,5**	**88,5**	83,5
Riešenie masívnych súprav sústav 3 lineárnych rovníc s 3 neznámymi	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,97	1,90	3,69	**98,5**	95	**92,25**
Lineárna algebra – Násobenie matíc typu 300 x 300	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,77	1,86	3,59	**88,5**	**93**	89,75

Tab. 19: Sumárne kvantitatívne charakteristiky praktických ukážok (pokračovanie)

Praktická ukážka	N_V			e (%)		
Paralelné grafické transformácie bitových máp (paralelizmus s vysokou úrovňou abstrakcie)	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	2,00	1,92	3,52	**100**	96	**88**
Paralelné grafické transformácie bitových máp (paralelizmus s nízkou úrovňou abstrakcie) – 10 bitmáp	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	2,05	1,99	3,28	**102,5**	99,5	**82**
Paralelné grafické transformácie bitových máp (paralelizmus s nízkou úrovňou abstrakcie) – 20 bitmáp	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,85	1,94	3,87	**92,5**	**97**	96,75
Vyhľadávanie prvočísel (riadený paralelizmus)	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,98	1,98	3,91	**99**	**99**	**97,75**
Vyhľadávanie prvočísel (natívny paralelizmus)	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,97	1,91	3,90	**98,5**	**95,5**	97,5
Numerická integrácia - natívny paralelizmus pomocou rozhrania OpenMP	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,99	2,00	3,98	**99,5**	**100**	**99,5**
Implicitný natívny paralelizmus – automatická paralelizácia algoritmu numerickej integrácie	AMD	IC2D	IC2Q	AMD	IC2D	IC2Q
	1,97	1,92	3,83	**98,5**	96	**95,75**

Legenda k tab. 19:

N_V	Nárast výkonnosti paralelného programu.
e	Efektivita využitia výpočtových zdrojov počítačového systému.
AMD	Procesor AMD Turion 64 X2.
IC2D	Procesor Intel Core 2 Duo T5800.
IC2Q	Procesor Intel Core 2 Quad Q6600.
e → max	Najefektívnejšie spracovanie paralelného programu.
e → min	Najmenej efektívne spracovanie paralelného programu.

11 Metodika riešiaca vývoj paralelných programov optimalizovaných pre beh na počítačových systémoch s viacjadrovými procesormi

Cieľom predkladanej metodiky je poskytnúť vývojárom, programátorom a softvérovým expertom súpravu postupov, usmernení a odporúčaní, ako čo možno najlepšie zvládnuť technicky náročný proces vývoja škálovateľných paralelných programov, ktoré dokážu využiť všetku disponibilnú výpočtovú kapacitu vysokovýkonných počítačových systémov s viacjadrovými procesormi. Metodika rieši 2 prístupy k tvorbe paralelných programov:

1. **Paralelizácia pôvodne sekvenčného programu.** V rámci tohto prístupu ponúka metodika hodnotné rady, ako postupovať pri paralelizácii existujúceho sekvenčného programu.

2. **Vývoj úplne nového paralelného programu.** V tomto prístupe sa metodika sústreďuje na analýzu, návrh a implementáciu nového paralelného programu, ktorý nepredstavuje paralelnú inováciu existujúceho sekvenčného programu.

11.1 Metodika riešiaca paralelizáciu pôvodne sekvenčného programu

Táto metodika je použiteľná pre softvérové firmy, ktoré už majú vytvorené portfólio sekvenčných programov, ktoré sú predmetom predaja na softvérovom trhu. Pri paralelizácii sekvenčných programov odporúčame postupovať nasledujúcim spôsobom:

1. **Stanovenie primárnych a sekundárnych cieľov, ktoré majú byť dosiahnuté prostredníctvom paralelizácie sekvenčného programu.** Primárnym cieľom paralelizácie je vždy škálovateľné zvýšenie výkonu pôvodne sekvenčného počítačového programu. Z tohto pohľadu je paralelizácia optimalizačnou technikou, ktorá generuje kvantifikovateľný nárast výkonnosti programu. Na tomto mieste si dovoľujeme poznamenať, že nárast výkonnosti programu musí byť škálovateľný. To znamená, že výkonnosť programu sa bude progresívne zvyšovať pri spracovaní programu na čoraz výkonnejších počítačových systémoch. V dobe prípravy tohto diela tvorili hlavný segment počítačov pre finálnych používateľov stroje s 2-, 3- a 4-jadrovými procesormi. Korektná paralelizácia však musí počítať nielen s aktuálnymi, ale aj budúcimi hardvérovými platformami. Po korektnej implementácii paralelizmu bude paralelný program preukazovať sústavný nárast svojej výkonnosti, a to bez ohľadu na to, na ako vyspelom stroji bude spracúvaný. Povedané inak, i dnešné paralelné programy musia byť schopné dosahovať škálovateľné nárasty výkonnosti pri behu na budúcich počítačových systémoch. K sekundárnym cieľom paralelizácie patrí najmä zlepšenie segmentácie programových modulov, algoritmická dekompozícia (čiže členenie algoritmov programu na sekvenčné a paralelné algoritmy) a lepšia správa paralelných algoritmov.

2. **Uskutočnenie výkonnostnej diagnostiky sekvenčného programu.** Odporúčame, aby sa pred samotným procesom paralelizácie sekvenčného programu uskutočnili diagnostické testy, ktoré preveria existujúcu výkonnosť tohto programu. Pri diagnostických testoch je potrebné zamerať

sa na kritické miesta programu. Kritické miesta programu sú tvorené množinou kritických algoritmov, ktoré spĺňajú nasledujúce vlastnosti:

1. Kritické algoritmy programu sú výpočtovo intenzívne.
2. Kritické algoritmy programu sú frekventovane spracúvané (volané).

Je zmysluplné, aby sme sa koncentrovali iba na kritické algoritmy sekvenčného programu, pretože práve pri nich existuje najväčšia pravdepodobnosť toho, že ich paralelizácia zabezpečí signifikantný nárast výkonnosti. Zameranie na výpočtovo intenzívne kritické algoritmy garantuje, že prínosy paralelizácie úplne prevýšia režijné náklady, ktoré sú s paralelizáciou spojené.

Z množiny kritických algoritmov sekvenčného programu, ktoré vyhovujú spomenutým kritériám, vyberieme tie, ktoré možno paralelizovať. Zostavíme teda podmnožinu paralelizovateľných kritických algoritmov sekvenčného programu. Je možné, že v procese analýzy kritických algoritmov zistíme, že nie všetky z nich sú vhodné na paralelizáciu. To sa stáva napríklad vtedy, keď je istý kritický algoritmus rýdzo sekvenčný – úlohy, ktoré realizuje, musia byť vykonané sériovo v presne stanovenom poradí. Aby mohla paralelizácia sekvenčného programu generovať maximálny možný nárast výkonnosti, je nutné minimalizovať absolútnu početnosť výskytu rýdzo sekvenčných kritických algoritmov. Ak budeme skúmať relatívne početnosti paralelizovateľných kritických algoritmov a rýdzo sekvenčných kritických algoritmov, tak môžeme konštatovať, že ich vzťah determinuje nárast výkonnosti sekvenčného programu plynúci z paralelizácie. Čím je relatívna početnosť paralelizovateľných kritických algoritmov vyššia, tým vyšší bude aj nárast výkonnosti generovaný paralelizáciou. A naopak, čím vyššia je relatívna početnosť rýdzo sekvenčných kritických algoritmov, tým menší efekt bude vyplývať z paralelizácie sekvenčného programu.

V tomto momente máme k dispozícii 2 podmnožiny kritických algoritmov sekvenčného programu: paralelizovateľné kritické algoritmy a rýdzo

sekvenčné kritické algoritmy. Zvýšenie výkonnosti 1. podmnožiny kritických algoritmov uskutočníme ich paralelizáciou – zhotovíme paralelné modely algoritmov, špecifikujeme druh paralelizmu, zvolíme jeho úroveň abstrakcie, a nakoniec prostredníctvom optimálnych techník paralelizmus implementujeme. Hoci výkonnosť 2. podmnožiny kritických algoritmov nezvýšime ich paralelizáciou, neznamená to, že nemôžeme uplatniť iné optimalizačné techniky. Výkon rýdzo sekvenčných kritických algoritmov dokážeme maximalizovať starostlivou analýzou ich tokov a použitím variabilných optimalizačných techník (napríklad reštrukturalizáciou kódu tak, aby sa znížili výkonnostné penalizácie generované lokalitou dát, či úpravou kódu tak, aby sa maximalizovala schopnosť procesora predpovedať budúce toky programových inštrukcií).

3. **Predikcia maximálneho možného nárastu výkonnosti sekvenčného programu, ktorý vyplýva z paralelizácie paralelizovateľných kritických algoritmov.** Na predikciu môžeme použiť Amdahlov zákon. Uvažujme nasledujúci modelový príklad:

    - Absolútna početnosť paralelizovateľných kritických algoritmov a rýdzo sekvenčných kritických algoritmov je 100.
    - Relatívna početnosť paralelizovateľných kritických algoritmov je 90 %.
    - Relatívna početnosť rýdzo sekvenčných kritických algoritmov je 10 %.
    - 1 paralelizovateľný kritický algoritmus vykonáva 50 elementárnych operácií, pričom čas spracovania jednej elementárnej operácie je 20 ms.
    - 1 rýdzo sekvenčný kritický algoritmus vykonáva 50 elementárnych operácií, pričom čas spracovania jednej elementárnej operácie je 20 ms.
    - Počet exekučných jadier viacjadrového procesora je variabilný.

Nárast výkonnosti sekvenčného programu generovaný paralelizáciou paralelizovateľných kritických algoritmov, bude podľa Amdahlovho zákona na počítači so 4-jadrovým procesorom nasledujúci:

$$N_V = \frac{T_S + T_P}{T_s + \frac{T_P}{n}} = \frac{1000 + 9000}{1000 + \frac{9000}{4}} = \frac{10000}{3250} = 3{,}08$$

Ak zdvojnásobíme počet exekučných jadier viacjadrového procesora, tak získavame priaznivejšiu kvantifikáciu nárastu výkonnosti sekvenčného programu po jeho paralelizácii:

$$N_V = \frac{T_S + T_P}{T_s + \frac{T_P}{n}} = \frac{1000 + 9000}{1000 + \frac{9000}{8}} = \frac{10000}{2125} = 4{,}71$$

Ak paralelný program spustíme na 16-jadrovom procesore, tak zaznamenáme ďalší dodatočný nárast výkonnosti:

$$N_V = \frac{T_S + T_P}{T_s + \frac{T_P}{n}} = \frac{1000 + 9000}{1000 + \frac{9000}{16}} = \frac{10000}{1562{,}5} = 6{,}4$$

Ak budeme počet exekučných jadier viacjadrového procesora ďalej zvyšovať, tak zistíme, že pre $n \to \infty$ dochádza k eliminácii člena $\frac{T_P}{n}$. To znamená, že maximálny možný teoretický nárast výkonnosti programu bude takýto:

$$N_V = \frac{T_S + T_P}{T_s} = \frac{10000}{1000} = 10$$

Záver uvažovaného modelového príkladu je nasledujúci: Relatívna početnosť rýdzo sekvenčných algoritmov vytvára obmedzenie nárastu výkonnosti paralelného programu, ktorý vznikne po paralelizácii pôvodne sekvenčného programu. V našom prípade je relatívna početnosť rýdzo sekvenčných algoritmov 10 %, čo predstavuje maximálne 10-násobný nárast výkonnosti.

4. **Implementácia stratégie maximalizácie nárastu výkonnosti paralelného programu.** Nami vytvorená stratégia zabezpečuje maximálny možný nárast výkonnosti paralelného programu prostredníctvom uskutočnenia 2 optimalizačných procesov:

    1. Paralelizácia paralelizovateľných kritických algoritmov sekvenčného programu.
    2. Optimalizácia rýdzo sekvenčných kritických algoritmov sekvenčného programu.

    Maximálny nárast výkonnosti paralelného programu bude zabezpečený vtedy, keď sa maximálne zvýši výkonnosť oboch charakterizovaných podmnožín algoritmov. Výkonnosť paralelizovateľných kritických algoritmov zvýšime ich paralelizáciou. Výkonnosť rýdzo sekvenčných kritických algoritmov zvýšime inými optimalizačnými technikami, ktoré nepočítajú so softvérovou paralelizáciou.

5. **Paralelizácia paralelizovateľných sekvenčných kritických algoritmov.** V tomto kroku je potrebné uskutočniť nasledujúce rozhodnutia:

    1. **Špecifikovať varianty paralelizmu.** Navrhujeme, aby sa výber realizoval podľa týchto odporúčaní (odporúčania sú uvedené v poradí ich priority):

        - **Implicitný paralelizmus a explicitný paralelizmus.** Ako sme aj sami overili v jednej z našich praktických ukážok, moderné prekladače sú schopné automaticky a implicitne paralelizovať sekvenčný zdrojový kód. Táto implicitná transformácia kódovej základne môže znamenať signifikantný nárast výkonnosti. Preto odporúčame, aby vývojári pred explicitnými zásahmi do zdrojového kódu sekvenčného programu využili schopnosti automatickej paralelizácie. Aspekty, ako sú hĺbka a rozsah automatickej

paralelizácie, môžu byť kontrolované prostredníctvom špeciálnych konfiguračných nastavení prekladača. (Tieto nastavenia determinujú stupeň agresivity prekladača pri realizácii automatickej paralelizácie.) Pre lepšiu prehľadnosť navrhujeme vždy zobraziť výslednú hodnotiacu štatistiku, v ktorej budú označené tie segmenty zdrojového kódu programu, ktoré boli podrobené automatickej paralelizácii. Rovnako radíme využiť diagnostické schopnosti prekladača pri uskutočnení automatickej vektorizácie zdrojového kódu sekvenčného programu.

- Ak implicitný paralelizmus neprináša očakávaný nárast výkonnosti programu, je potrebné implementovať explicitný paralelizmus. Pri výbere explicitného paralelizmu sami určíme, ktoré segmenty zdrojového kódu budú pôsobiť ako paralelné regióny a navrhneme najlepšiu stratégiu ich následnej paralelizácie.

- **Dátový paralelizmus, úlohový paralelizmus a paralelizmus dátových tokov.** V tomto smere sa snažíme identifikovať dáta a dátové štruktúry, s ktorými paralelizovateľné sekvenčné kritické algoritmy pracujú. Rovnako uskutočňujeme analýzu nezávislých tokov programových inštrukcií, ktoré môžeme zapuzdriť do diskrétnych paralelizovateľných úloh. Pri aplikácii dátového a úlohového paralelizmu vždy zostavujeme pamäťové modely dátových entít, ktoré v analyzovaných procesoch vystupujú, a identifikujeme potenciálne vzájomné interakčné väzby, ktoré medzi týmito entitami existujú. Ak je nevyhnutný súbežný prístup k zdieľaným dátovým entitám, ako najlepšie riešenie sa javí privatizovať tieto dátové entity pre každé programové vlákno, ktoré s nimi musí manipulovať. Po privatizácii už dátové entity nebudú

zdieľané, ale súkromné. V prípade, keď je privatizácia nerealizovateľná, selektujeme synchronizačné objekty, ktorými zabezpečíme vzájomne vylučujúci sa prístup k dátovým entitám. V záujme minimalizácie výkonnostných penalizácií, ktoré sa viažu na proces synchronizácie, aplikujeme atomické operácie, monitory a mutexy.

- **Deklaratívny paralelizmus a imperatívny paralelizmus.** Keďže takmer všetky programovacie jazyky s najvyššími trhovými penetráciami patria k hybridným imperatívnym objektovo orientovaným jazykom, ako najvhodnejšia alternatíva sa javí použitie imperatívneho paralelizmu. Vďaka stratégii koevolúcie imperatívnych a deklaratívnych programovacích jazykov sú mnohé prvky deklaratívneho, resp. funkcionálneho programovania zavádzané aj do imperatívnych programovacích prostriedkov. Ak je pri paralelizácii sekvenčného programu použitý imperatívny programovací jazyk s podporou vybranej podmnožiny rysov deklaratívneho programovania, tak jednoznačne odporúčame tieto rysy pri paralelizácii zužitkovať.

2. **Určiť úroveň abstrakcie paralelizmu.** Čím vyššia úroveň paralelizmu bude zvolená, tým vyššiu pracovnú produktivitu vývojárov môžeme očakávať. Preto radíme pracovať s vysoko abstraktným paralelizmom. Tak sa efektívne vyhneme nutnosti vykonávať vo vlastnej réžii všetky parciálne činnosti, ktoré súvisia s manažovaním životných cyklov pracovných vlákien paralelného programu.

3. **Zvoliť optimálne nástroje na implementáciu paralelizmu.** Paralelizácia paralelizovateľných kritických algoritmov sekvenčného programu bude najrýchlejšia vtedy, keď sa použijú najvhodnejšie nástroje. K nim patrí:

- **Hybridný objektovo orientovaný imperatívny programovací jazyk s podporou množiny syntakticko-sémantických rysov funkcionálneho programovania.** Pre vývoj riadených programov, ktoré budú spracúvané na platforme Microsoft .NET Framework 4.0, možno použiť ľubovoľný z dostupných .NET-kompatibilných programovacích jazykov. Odporúčame, aby si vývojári vybrali ten jazyk, ktorý najlepšie poznajú a s ktorým majú najväčšie praktické skúsenosti.

- **Paralelná platforma – platforma podporujúca paralelizáciu výpočtových procesov.** Pri vývoji riadených programov zacielených na platformu Microsoft .NET Framework 4.0 navrhujeme využiť knižnicu Task Parallel Library (TPL). Knižnica TPL je prístupná z ktoréhokoľvek .NET-kompatibilného programovacieho jazyka (C# 4.0, Visual Basic 2010, F# a C++/CLI). Pri vývoji natívnych programov pripadá do úvahy viacero paralelných platforiem, ktorých jadrová funkcionalita je zapuzdrená do knižníc. Riešenie od spoločnosti Microsoft má názov Parallel Patterns Library (PPL) a je použiteľné z jazyka C++. Knižnica PPL svojím syntakticko-sémantickým modelom stavia na štandardnej šablónovej knižnici jazyka C++ (STL). Pritom využíva nové prvky pripravovaného ISO štandardu jazyka C++ (v čase tvorby tohto diela s pracovným označením C++0x), ku ktorým patria napríklad λ-výrazy a mechanizmus typovej inferencie. Riešenie od spoločnosti Intel sa volá Intel Threading Building Blocks (TBB) a je rovnako založené na štandardnej šablónovej knižnici jazyka C++. Otvoreným riešením je rozhranie OpenMP dodávané dovedna s exekučným prostredím. Paralelizmus je v OpenMP implementovaný predovšetkým pomocou direktív, ktoré usmerňujú prácu prekladačov programovacích jazykov C, C++ a Fortran. Pre jazyk C++ sú významné najmä 2 prekladače: Visual C++ 2010 (podporuje

rozhranie OpenMP verzie 2.0) a Intel C++ Compiler 11.1 (podporuje rozhranie OpenMP verzie 3.0).

- **Integrované vývojové prostredie so súpravou návrhárskych, implementačných, diagnostických a monitorovacích nástrojov umožňujúcich komfortnú tvorbu paralelných programov.** Z našej analýzy vyplýva, že najväčšiu pridanú hodnotu pri vývoji riadených a natívnych paralelných programov generuje kombinácia produktov Microsoft Visual Studio 2010 Team System a Intel Parallel Studio. Nástroje, ktoré sú zahrnuté v zmienených produktoch, poskytujú komplexnú podporu pri analýze, návrhu, implementácii, testovaní a nasadení robustných paralelných programov.

6. **Optimalizácia rýdzo sekvenčných kritických algoritmov.** Vzhľadom na to, že pri rýdzo sekvenčných kritických algoritmoch nie sme schopní dosiahnuť nárast výkonnosti paralelizáciou, musíme nasadiť iné optimalizačné techniky, napríklad:

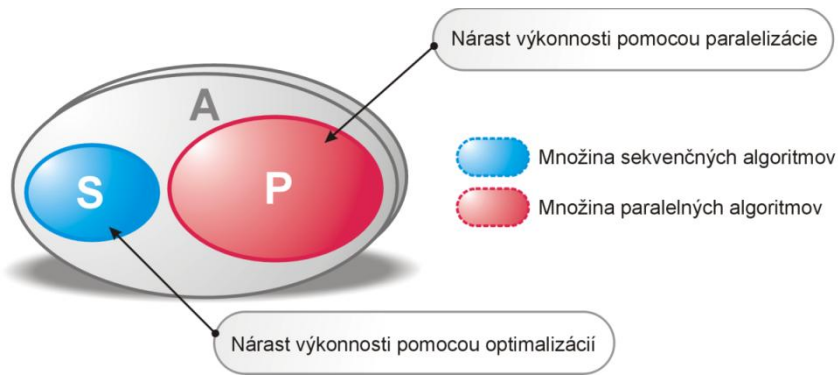
   - optimalizácia práce s rýchlymi vyrovnávacími pamäťami procesora,
   - optimalizácia pamäťového zarovnania dátových entít,
   - reštrukturalizácia zdrojového kódu s cieľom efektívnych interprocedurálnych optimalizácií,
   - reštrukturalizácia zdrojového kódu s cieľom využitia bázových hardvérových technológií procesora,
   - transformácia mimoriadne kritických segmentov zdrojového kódu do nízkoúrovňového kódu jazyka symbolických inštrukcií.

7. **Kvantifikácia nárastu výkonnosti paralelného programu po implementácii stratégie maximalizácie nárastu výkonnosti.** Po implementácii stratégie maximalizácie nárastu výkonnosti sme schopní určiť celkový nárast výkonnosti paralelného programu:

$$C_{NV} = N_P + N_O$$

kde:

- C_{NV} je celkový nárast výkonnosti po paralelizácii sekvenčného programu.
- N_P je nárast výkonnosti plynúci z paralelizácie paralelizovateľných kritických algoritmov sekvenčného programu.
- N_O je nárast výkonnosti plynúci z optimalizácie rýdzo sekvenčných kritických algoritmov sekvenčného programu.



Obr. 33: Stratégia maximalizácie nárastu výkonnosti programu

8. **Verifikácia korektnosti a ladenie paralelného programu.** Po implementácii paralelizmu do pôvodne sekvenčného programu je potrebné otestovať, či pretransformovaný paralelný program pracuje správne. Primárne je nutné overiť, či paralelný program vykonáva svoju činnosť deterministicky a či produkuje korektné výstupy. V závislosti od zložitosti paralelného programu odporúčame spracovať automatizovanú kolekciu testov, ktoré preveria, či sa v paralelnom programe nevyskytujú skryté chyby, kolízne stavy či iné nepriaznivé artefakty, ktoré by limitovali výkon programu. Ak paralelný program po svojom spustení generuje rádovo nižší nárast výkonnosti, než aký by sme očakávali, tak s najväčšou pravdepodobnosťou obsahuje niektoré z kritických latentných chýb, ku

ktorým patria hlavne preteky vlákien, uviaznutia a narušenia pamäťových lokalít. Príčinou nízkeho nárastu výkonnosti však môže byť aj nerovnomerná distribúcia pracovného zaťaženia pracovných vlákien paralelného programu. S týmto problémom sa často stretávame vtedy, keď vývojári uprednostnia implementáciu explicitného paralelizmu s nízkou úrovňou abstrakcie. Riešením je navrhnutie sofistikovanejšieho algoritmu, ktorý bude generovať rovnomerné zaťaženie pracovných vlákien paralelného programu. Aj preto, aby sa takýto druh výkonnostných degradácií neobjavoval, odporúčame vždy implementovať vysoko abstraktný paralelizmus. Ten dokáže pomocou inteligentnej techniky transportu úloh naprieč viacerými pracovnými vláknami garantovať ich optimálne zaťaženie.

9. **Experimentálne nasadenie paralelného programu.** Výkonnosť paralelného programu je užitočné otestovať ešte pred ostrým nasadením na množine variabilne výkonných počítačových systémov, ktoré sú vybavené 1-prvkovou množinou viacjadrových procesorov, respektíve n-prvkovou množinou viacjadrových procesorov. Testy odporúčame realizovať vždy s adekvátnou súpravou vstupných dát, resp. s adekvátnym počtom úloh, ktoré bude paralelný program spracúvať, resp. vykonávať pri svojej typickej exekúcii u finálneho používateľa. Pre dôkladnejšiu analýzu nárastu výkonnosti paralelného programu navrhujeme uskutočniť nasledujúcu kolekciu testov, a to v 3 výkonnostných režimoch (úsporný režim, rovnovážny režim a vysokovýkonný režim[5]):

    - **Spustiť a otestovať paralelný program v ideálnom stave.** V operačnom systéme sú spustené len systémové procesy, žiadny iný používateľský program nebeží.

    - **Spustiť a otestovať paralelný program pri bežnom zaťažení počítačového systému.** V operačnom systéme sú spustené

[5] Selekcia výkonnostných režimov sa v operačných systémoch Microsoft Windows Vista a Microsoft Windows 7 uskutočňuje v Ovládacích paneloch.

systémové procesy a n používateľských programov, ktoré alokujú 30 % – 70 % dostupného voľného pamäťového priestoru.

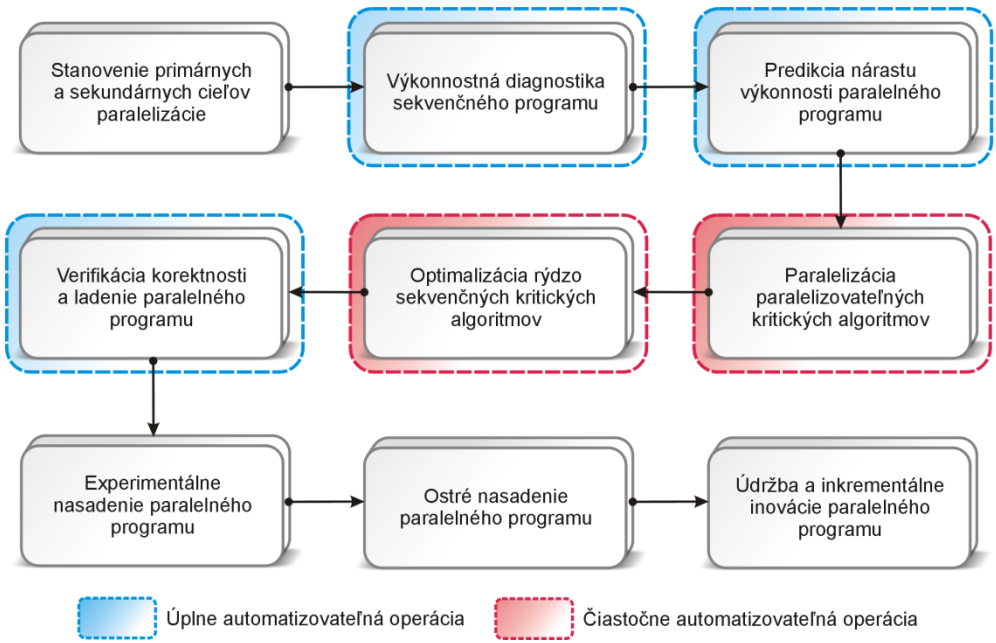
- **Spustiť a otestovať paralelný program pri vysokom zaťažení počítačového systému.** V operačnom systéme sú spustené systémové procesy a n používateľských programov, ktoré alokujú viac ako 70 %, no menej ako 90 % dostupného voľného pamäťového priestoru.

- **Spustiť a otestovať paralelný program pri maximálnom zaťažení počítačového systému.** V operačnom systéme sú spustené systémové procesy a n používateľských programov, ktoré alokujú viac ako 90 % dostupného voľného pamäťového priestoru.

10. **Ostré nasadenie paralelného programu.** Po svojej finalizácii je paralelný program pripravený na distribúciu finálnym používateľom. Vďaka škálovateľnosti paralelného programu je maximalizovaná jeho schopnosť prispôsobiť sa variabilne výkonnému počítačového systému a využiť všetku jeho disponibilnú výpočtovú kapacitu.

11. **Riadenie zmien, údržba a inkrementálne inovácie paralelného programu.** Podobne ako každý iný softvérový produkt, aj paralelný program bude potrebné udržiavať v optimálnej kondícii pomocou inkrementálnych inovácií, ktoré budú zvyšovať subjektívnu spokojnosť používateľa s programom.

Grafický model stvárňujúci hlavné etapy metodiky, ktorá rieši paralelizáciu existujúceho sekvenčného programu predstavuje obr. 34.



Obr. 34: Vizualizácia hlavných bodov metodiky
pre paralelizáciu existujúceho sekvenčného programu

11.2 Metodika riešiaca vývoj úplne nového paralelného programu

Pri vývoji nového paralelného programu je nutné venovať zvlášť dôkladnú pozornosť analýze a návrhu, pretože ich korektné spracovanie následne vymedzuje možnosti implementácie paralelného programu.

Pri vývoji nového paralelného programu odporúčame postupovať takto:

1. Vytvoriť dátový a objektový model paralelného programu.

2. Analyzovať inštancie dátových štruktúr, ktoré umožnia úplné paralelné spracovanie manipulačných operácií.

3. Analyzovať inštancie dátových štruktúr, ktoré umožnia čiastočné paralelné spracovanie manipulačných operácií s využitím synchronizácie.

4. Analyzovať procesy paralelného programu, ktoré môžu byť úplne vykonávané paralelne.

5. Analyzovať procesy paralelného programu, ktoré môžu byť čiastočne vykonávané paralelne s využitím synchronizácie.

6. Analyzovať procesy paralelného programu, ktoré nemôžu byť vykonávané paralelne – tzv. rýdzo sekvenčné procesy.

7. Špecifikovať paralelné algoritmy, ktoré budú aplikovať dátový paralelizmus v súvislosti s inštanciami dátových štruktúr analyzovanými v 2. a 3. bode tejto metodiky.

8. Špecifikovať paralelné algoritmy, ktoré budú aplikovať úlohový paralelizmus v súvislosti s procesmi analyzovanými v 4. a 5. bode tejto metodiky.

9. Špecifikovať paralelné algoritmy, ktoré budú aplikovať paralelizmus dátových tokov v súvislosti s inštanciami dátových štruktúr analyzovanými v 2. a 3. bode tejto metodiky a procesmi analyzovanými v 4. a 5. bode tejto metodiky.

10. Špecifikovať kritické paralelné algoritmy paralelného programu.

11. Špecifikovať rýdzo sekvenčné algoritmy paralelného programu.

12. Špecifikovať kritické rýdzo sekvenčné algoritmy paralelného programu.

13. Uskutočniť predikciu maximálneho možného nárastu výkonnosti paralelného programu, ktorý vyplýva z paralelizácie paralelizovateľných kritických algoritmov.

14. Implementovať stratégiu maximalizácie nárastu výkonnosti paralelného programu:

    - Paralelizovať paralelizovateľné kritické algoritmy sekvenčného programu.
    - Optimalizovať rýdzo sekvenčné kritické algoritmy sekvenčného programu.

15. Paralelizovať kritické paralelné algoritmy:

    1. Špecifikovať varianty paralelizmu:

        - Implicitný paralelizmus a explicitný paralelizmus.
        - Dátový paralelizmus, úlohový paralelizmus a paralelizmus dátových tokov.
        - Deklaratívny paralelizmus a imperatívny paralelizmus.

    2. Určiť úroveň abstrakcie paralelizmu:

        - Paralelizmus s nízkou úrovňou abstrakcie.
        - Paralelizmus so strednou úrovňou abstrakcie.
        - Paralelizmus s vysokou úrovňou abstrakcie.

    3. Zvoliť optimálne nástroje na implementáciu paralelizmu:

        - Hybridný objektovo orientovaný imperatívny programovací jazyk s podporou množiny syntakticko-sémantických rysov funkcionálneho programovania.

- Paralelná platforma – platforma podporujúca paralelizáciu výpočtových procesov.

- Integrované vývojové prostredie so súpravou návrhárskych, implementačných, diagnostických a monitorovacích nástrojov umožňujúcich komfortnú tvorbu paralelných programov.

16. Optimalizovať rýdzo sekvenčné kritické algoritmy.

17. Kvantifikovať nárast výkonnosti paralelného programu po implementácii stratégie maximalizácie nárastu výkonnosti.

18. Verifikovať korektnosť paralelného programu a odladiť ho.

19. Experimentálne nasadiť paralelný program.

20. Uskutočniť ostré nasadenie paralelného programu.

21. Riadiť zmeny, udržiavať a inkrementálne inovovať paralelný program.

Záver

Vážené čitateľky, vážení čitatelia,

sme nesmierne radi, že ste spoločne s nami absolvovali základný kurz praktického paralelného programovania v jazykoch C# 4.0 a C++. Sme si vedomí toho, že táto kniha podáva iba ochutnávku všetkých magických aspektov praktického paralelného programovania. Ak budete mať chuť sa aj naďalej v tomto turbulentnom odbore softvérového inžinierstva vzdelávať, dovoľujeme si na záver pripojiť viacero hodnotných zdrojov technickej literatúry, ktoré dokážu ďalej zvyšovať hodnotu vášho ľudského kapitálu:

1. AKHTER, S., ROBERTS, J.: **Multi-core Programming**. Hillsboro: Intel Press, 2006, ISBN 0-9764832-4-6.
2. DONGARRA, J., FOSTER, I., FOX, G., GROPP, W., KENNEDY, K., TORCZON, L., WHITE, A.: **Sourcebook of Parallel Computing**. San Francisco: Morgan Kaufmann Publishers, 2003, ISBN 978-1-55860-871-9.
3. GERBER, R., BIK, A. C. J., SMITH, K. B., TIAN, X.: **The Software Optimization Cookbook**. Hillsboro: Intel Press, 2006, ISBN 0-9764832-1-1.
4. RAJASEKARAN, S., REIF, J.: **Handbook of Parallel Computing (Models, Algorithms and Applications)**. Chapman & Hall/CRC: Boca Raton, Florida, 2008, ISBN 978-1-58488-623-5.
5. BRESHEARS, C.: **The Art of Concurrency**. Sebastopol: O'Reilly, 2009, ISBN 978-0-596-52153-0.
6. HUGHES, C., HUGHES, T.: **Professional Multicore Programming**. Indianapolis: Wiley Publishing, 2008, ISBN 978-0-470-28962-4.
7. DUFFY, J.: **Concurrent Programming on Windows**. Boston: Addison-Wesley Pearson Education, 2009, ISBN 978-0-321-43482-1.

Želáme vám veľa úspechov pri analýze, návrhu a implementácii paralelných programov!

Autor a realizačný tím

O autorovi

Ing. Ján Hanák, MVP, vyštudoval Ekonomickú univerzitu v Bratislave. Tu, na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky (KAI FHI), pracuje ako vysokoškolský pedagóg. Prednáša a vedie semináre týkajúce sa programovania a vývoja počítačového softvéru v programovacích jazykoch C, C++ a C#. Okrem spomenutej trojice jazykov patrí k jeho obľúbeným programovacím prostriedkom tiež Visual Basic, F# a C++/CLI.

Je nadšeným autorom odbornej počítačovej literatúry. V jeho portfóliu môžete nájsť nasledujúce knižné tituly:

1. **Programování v jazyce C**. Kralice na Hané: Computer Media, 2009.
2. **Praktické paralelné programovanie v jazykoch C# 4.0 a C++**. Brno: Artax, 2009.
3. **C++/CLI – Praktické príklady**. Brno: Artax, 2009.
4. **C# 3.0 – Programování na platformě .NET 3.5**. Brno: Zoner Press, 2009.
5. **C++/CLI – Začínáme programovat**. Brno: Artax, 2009.
6. **C#: Akademický výučbový kurz**. Bratislava: Vydavateľstvo EKONÓM, 2009.
7. **Základy paralelného programovania v jazyku C# 3.0**. Brno: Artax, 2009.
8. **Objektovo orientované programovanie v jazyku C# 3.0**. Brno: Artax, 2008.
9. **Inovácie v jazyku Visual Basic 2008**. Praha: Microsoft, 2008.
10. **Visual Basic 2008: Grafické transformácie a ich optimalizácie**. Bratislava: Microsoft Slovakia, 2008.
11. **Programovanie B – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C++)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
12. **Programovanie A – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
13. **Expanzívne šablóny: Príručka pre tvorbu "code snippets" pre Visual Studio**. Bratislava: Microsoft Slovakia, 2008.
14. **Kryptografia: Príručka pre praktické odskúšanie symetrického šifrovania v .NET Framework-u**. Bratislava: Microsoft Slovakia, 2007.

15. **Príručka pre praktické odskúšanie vývoja nad Windows Mobile 6.0**. Bratislava: Microsoft Slovakia, 2007.
16. **Príručka pre praktické odskúšanie vývoja nad DirectX**. Bratislava: Microsoft Slovakia, 2007.
17. **Príručka pre praktické odskúšanie automatizácie aplikácií Microsoft Office 2007**. Bratislava: Microsoft Slovakia, 2007.
18. **Visual Basic 2005 pro pokročilé**. Brno: Zoner Press, 2006.
19. **C# – praktické příklady**. Praha: Grada Publishing, 2006.
20. **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Praha: Microsoft, 2006.
21. **Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005**. Praha: Microsoft, 2005.
22. **Visual Basic .NET 2003 – Začínáme programovat**. Praha: Grada Publishing, 2004.

V rokoch 2006 – 2009 bol jeho prínos vývojárskym komunitám ocenený celosvetovými vývojárskymi titulmi **Microsoft Most Valuable Professional (MVP)** s kompetenciou **Visual Developer – Visual C++**.

Ocenenia:

1. Spoločnosť Microsoft ČR udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za napísanie prvej vysokoškolskej učebnice „C++/CLI – Začíname programovať“ o algoritmizácii a programovaní v jazyku C++/CLI.

2. Spoločnosť Microsoft Slovakia udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za nasadenie najnovších vývojárskych technológií do výučbového procesu v publikácii „Základy paralelného programovania v jazyku C# 3.0“.

3. Spoločnosť Microsoft ČR udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za mimoriadne úspešné odborné knižné publikácie „Objektovo orientované programovanie v jazyku C# 3.0“ a „Inovácie v jazyku Visual Basic 2008“.

4. Spoločnosť Microsoft Slovakia udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za zlepšovanie akademického ekosystému a za signifikantné rozširovanie technológií a programovacích jazykov spoločnosti Microsoft na akademickej pôde.

5. Spoločnosť Grada Publishing udelila knihe „C# – praktické příklady" Ing. Jána Hanáka v roku 2006 ocenenie „Najúspešnejšia novinka vydavateľstva Grada v oblasti programovania za rok 2006".

Kontakt s vývojármi a programátormi udržiava najmä prostredníctvom technických seminárov a odborných konferencií, na ktorých vystupuje. Za všetky vyberáme tieto:

- Technický seminár **Paralelné programovanie**. KAI FHI EU a Microsoft Slovakia. Bratislava 21. 10. 2009.

- Konferencia **Software Developer 2007**, príspevok na tému „Představení produktu Visual C++ 2005 a jazyka C++/CLI". Praha 19. 6. 2007.

- Technický seminár **Novinky vo Visual C++ 2005**. Microsoft Slovakia. Bratislava 3. 10. 2006.

- Technický seminár **Visual Basic 2005 a jeho cesta k Windows Vista**. Microsoft Slovakia. Bratislava 27. 4. 2006.

Ako autor má mnohoročné skúsenosti s prácou v elektronických a printových médiách. Počas svojej kariéry pôsobil ako odborný autor alebo odborný redaktor v nasledujúcich časopisoch:

- PC WORLD,
- SOFTWARE DEVELOPER,
- CONNECT!,
- COMPUTERWORLD,
- INFOWARE,
- PC REVUE

- a CHIP.

Dovedna publikoval viac ako 250 odborných a populárnych prác venovaných vývoju počítačového softvéru.

Akademický blog autora môžete sledovať na adrese: http://blog.aspnet.sk/hanja/.

Ak sa chcete s autorom spojiť, píšte na hanja@stonline.sk.

Ing. Ján Hanák, MVP, je najcennejším odborníkom spoločnosti Microsoft s kompetenciou Visual Developer – Visual C++. Je autorom 22 odborných kníh, príručiek a praktických cvičení o programovaní a vývoji počítačového softvéru. Pracuje ako vysokoškolský pedagóg na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky Ekonomickej univerzity v Bratislave. Prednáša a vedie semináre z programovania v jazykoch C, C++ a C#.
V rámci svojej vedeckej činnosti sa zaoberá problematikou štruktúrovaného, objektovo orientovaného, komponentového, funkcionálneho a paralelného programovania.

ISBN: 978-80-87017-06-7