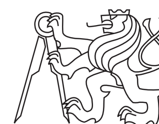
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Simulace pohybu těles v planetárním systému

Filip Krutil

Vedoucí práce: Ing. Ivan Šimeček, PhD.

18. února 2016

Prohlášení

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

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

V Praze dne 18. února 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Filip Krutil. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Krutil, Filip. *Simulace pohybu těles v planetárním systému*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce je zaměřená na optimalizaci kódu, který představuje simulaci pohybu těles, které mezi sebou gravitačne interagují, konkrétně byla k simulaci použita data představující sluneční soustavu. Neoptimalizovaný algoritmus byl napsán na základě Newtonova gravitačního zákona v jazyce C++ a poté optimalizován pomocí různých nastavení kompilátoru GCC, malými úpravami zdrojového kódu, ale i jeho rozsáhlejšími transformacemi jako je rozbalování cyklů, nebo způsob uložení dat v paměti. V neposlední řadě byl pak algoritmus paralelizován pro systémy s mnoha výpočetními jádry pro paralelní výpočty technologií OpenMP. Z přiložených zdrojových kódů je pak možné přeložit různé verze programu, podle použitých optimalizací od neoptimalizovaného až po nejrychlejší paralelní verzi programu pro 24 výpočetních vláken. Spuštěním takto přeloženého programu se vstupními daty představujícími stav sluneční soustavy v určitém čase, lze získat stav soustavy odpovídající stavu po uběhnutí určitého předem zvoleného času a tento vizualizovat v podobě 3D grafu vytvořeném pomocí nástroje Gnuplot.

Klíčová slova C/C++, simulace pohybu planet, optimalizace, rozbalování cyklů, paralelizace, OpenMP

Abstract

This thesis is focused on optimization of code representing celestial bodies movements which interacts by gravitational forces. In this case data representing Solar system was used for the simulation. Unoptimized algorithm is basicaly Newton's law of universal gravitation written in the C++ programming language which was then optimized by using different GCC compiler options, making small changes in the code, and also more extensive alterations such as cycle unrolling or the way how data was stored in the memory. Last but not least the algorithm was adapted for multicore systems for parallel computing by the OpenMP technology. It is possible to compile several versions of differently optimized or adjusted programs from attached source codes starting with a basic version and ending with the fastest 24 threads parallel version. By running this program with data representing Solar System in given time, it is possible to get a new system state after certain time given beforhand and vizualize the system by the Gnuplot tool in the form of a 3D graph.

Keywords C/C++, planetary movement simulation, optimization, cycle unrolling, parallelization, OpenMP

Obsah

Úvod **1**

1 Cíl práce **3**
1.1 Přesnost simulace 3

2 Teoretická část **5**
2.1 Problém N těles 5
2.2 Odvození použitých rovnic 5
2.3 Simulace 6
2.4 Podobné programy 6
2.5 Techniky optimalizace 7

3 Realizace **11**
3.1 Popis použitého neoptimalizovaného algoritmu 11
3.2 Použití programu a jeho funkcionalita 13

4 Měření výkonnosti **15**
4.1 Výkonnost neoptimalizovaného algoritmu 15
4.2 Optimalizované verze 16

Závěr **27**

Literatura **29**

A Seznam použitých zkratek **31**

B Obsah přiloženého CD **33**

C Tabulky naměřených výkonností **35**

Seznam obrázků

3.1 Odchylky pro různé časové kroky 13

4.1 Výkonnost neoptimalizovaného algoritmu 16
4.2 Výkonnost neoptimalizovaného algoritmu 16
4.3 Výkonnost pro různé nastavení kompilátoru 17
4.4 Srovnání výkonnosti kódu s podmínkou a bez 18
4.5 Výkonnost pro různé nastavení kompilátoru 19
4.6 Srovnání nerozbaleného kódu s rozbalováním pro různé faktory rozbalení 20
4.7 Srovnání kódu s kombinovaným faktorem rozbalení 21
4.8 Srovnání nerozbaleného kódu bez a s volbou -funroll-loops 21
4.9 Srovnání dvakrát rozbaleného kódu bez a s volbou -funroll-loops . 22
4.10 Srovnání čtyřikrát plus dvakrát rozbaleného kódu bez a s volbou -funroll-loops 22
4.11 Srovnání způsobů uložení dat v paměti metodami AOS a SOA . . 23
4.12 Srovnání vektorizace s nevektorizovanou verzí 24
4.13 Srovnání různých verzí kódu paralelizovaných 4 vlákny 25
4.14 Srovnání zrychlení pro různé počty vláken 26

Úvod

Tato práce implementuje algoritmus pro simulaci pohybu těles, konkréně planet ve sluneční soustavě. Nemůže však soupeřit s ústavy zabývajícími se vesmírem a jejich superpočítači, takže je poté zaměřena na optimalizaci a paralelizaci tohoto kódu.

V Teoretické části se zabývá problémem N-těles, který se týká této problematiky. Tento je poté vyřešen metodou přímé simulace a implementací Newtonova gravitačního zákona.

Dále jsou v kapitole Techniky optimalizace rozebrány některé možnosti zrychlení a úprav zdrojového kódu a technologie OpenMP, která je použita pro paralelizaci kódu.

Výkonnosti všech verzí kódu jsou poté měřeny a graficky znázorněny v kapitole Měření výkonnosti.

Kapitola 1

Cíl práce

Práce si neklade za cíl simulovat vědecky přesný pohyb těles v planetarním systému, ale implementaci časově relativně náročného kódu (zde se jedná o kvadratickou časovou složitost) na základě fyzikálního problému, který poté bude postupně zoptimalizován různými optimalizačními technikami jak pro sekvenční výpočet, tak i vhodné rozložení problému mezi mnoho výpočetních vláken na paralelním systému. Tyto optimalizační techniky pak nejsou nijak závislé na tomto konkrétním kódu a lze je využít v jiných podobných případech.

1.1 Přesnost simulace

Algoritmus pro simulaci v této práci dále popsaný je pouze aproximací zohledňující pouze gravitační síly mezi tělesy a to jednak z důvodu diskretizace jinak spojitého systému, který se tak stává závislým na časovém kroku simulace, kde s každým krokem dochází ke kumulaci chyby, ale i způsobem, jakým jsou ukládána reálná čísla v paměti počítače, kde často dochází k zaokrouhlovacím chybám.

Kapitola 2

Teoretická část

2.1 Problém N těles

Klasický problém N těles představuje pohyb N těles (částic, planet, ...) v prostoru, kde dochází k interakcím mezi těmito tělesy. Algoritmus řešící tento problém, má mnohé využití například v astrofyzice, k simulaci pohybu planet, hvězd a dalších nebeských objektů, ale i v molekulární dynamice, pro simulaci jednotlivých molekul [1].

V případě simulace sluneční soustavy se tedy jedná o nebeská tělesa (Slunce, planety, měsíce, atd.), které na sebe vzájemně (každý s každým) působí gravitační silou. Velikost těchto sil určíme pomocí Newtonova gravitačního zákona (kapitola 2.2.1).

2.2 Odvození použitých rovnic

2.2.1 Newtonův gravitační zákon

Podle Newtonova gravitačního zákona na sebe každé dvě tělesa o hmotnostech m_i a m_j působí gravitační silou F_g přímo úměrnou hmotnostem těles a nepřímo úměrnou čtverci jejich vzdálenosti [2].

$$F_g = G\frac{m_i m_j}{r^2}$$

Kde G je gravitační konstanta, $G = 6.674*10^{-11} Nm^2kg^{-2}$, m_i a m_j hmotnosti těles a r vzdálenost středů obou těles.

Vektorově lze pak sílu, kterou působí těleso i na těleso j zapsat takto.

$$\vec{F_{ij}} = -G\frac{m_i m_j}{r^2}\frac{\mathbf{r}}{r}$$

Kde $\mathbf{r}$ je polohový vektor prvního tělesa vzhledem ke druhému.

2.2.2 Suma všech sil

Pro potřeby simulace je pak nutné spočítat sumu všech sil působících na každé těleso.

$$\vec{F_j} = \sum_{j=1,i\neq j}^{N} G\frac{m_i m_j}{r^2}\frac{\mathbf{r}}{r} = \sum_{j=1,i\neq j}^{N} \frac{Gm_i m_j(r_j - r_i)}{(\sqrt{((j_x - i_x)^2 + (j_y - i_y)^2 + (j_z - i_z)^2)}^3}$$

2.3 Simulace

2.3.1 Přímá

Pokud známe vektory sil působících na každé těleso, můžeme vypočítat jejich výslednou polohu a rychlost danou působením těchto sil po krátký časový úsek *dt*. Takovouto diskretizací pohybu, kdy po *dt* je těleso v rovnoměrně zrychleném přímočarém pohybu, však dochází k nepřesnostem v simulaci, proto je nutné zvolit *dt* vhodně dlouhý (více v kapitole 3.1.1). Takový způsob simulace se nazývá přímý (Direct, Particle-Particle) [3].

2.3.2 Jiné metody simulace

Přímá metoda simulace je vhodná pro počet těles v řádu tisíců, tedy i pro sluneční soustavu. Pro vyšší počty těles by bylo nutné použít některou z méně přesných, ale rychlejších metod, jako je například algoritmus Barnes-Hut [4], těmito se ale tato práce nezabývá.

2.3.3 Zdrojová data

Simulace vychází z daného stavu soustavy v konkrétním čase. Tím se rozumí textový soubor, který obsahuje data o simulovaných tělesech. Tato data jsou získána z webových stránek NASA [5] v následujícím formátu. Středem soustavy je hvězda, tedy Slunce, které představuje střed, tedy bod (0,0,0), který zůstává středem po celou dobu simulace. Dále u každého tělesa potřebujeme označení (název), polohu vzhledem ke středu (Slunci), kartézské souřadnice x, y, z. Jednotkou souřadnic je metr. Jako další je zapotřebí vektor rychlosti vx, vy, vz. Jednotkou rychlosti je metr za sekundu. A jako poslední je potřeba znát hmotnost tělesa m v kilogramech.

2.4 Podobné programy

Existuje mnoho programů, které nějakým způsobem simulují pohyb ve sluneční soustavě, ať už se jedná o výukové programy, animace, nebo programy, které generují údaje o tělesech v daném čase v textové podobě.

Pro srovnání s touto prací se nabízí bakalářská práce Simulátor sluneční soustavy z ČVUT FEL Z roku 2008 [6] a na ní navazující diplomová práce Pokročilý simulátor sluneční soustavy z roku 2010 [7]. Obě práce mají za cíl vytvořit program simulující pohyb vesmírných těles ve Sluneční soustavě v jazyce Java. V prvním případě pomocí matematického modelu a ve druhém pomocí fyzikálního modelu. Matemetický model zohledňuje Kepplerovy zákony a z něj odvozené pohybové rovnice na základě pozorování těles, kde každé těleso lze popsat takzvanými orbitálními elementy jako jsou hlavní poloosa, excentricita, argument délky pericentra, délka vzestupného uzlu, inklinace a střední anomálie, které jednoznačně stanoví rozměr, natočení a pozici tělesa na elipse, po které se pohybuje. Druhá práce pracuje s přesnější Hamiltonovou mechanikou a z ní odvozenými rovnicemi a podobně jako v této práci s numerickou metodou představující aproximaci diskretizací spojitého reálného světa.

Tato práce oproti výše zmíněným pracuje s klasickou Newtonovou mechanikou. Příčina pohybu je tedy pouze interakce mezi tělesy způsobená gravitačními silami. Hlavním cílem pak není výstup v podobě animace pohybu soustavy, ale optimalizace kódu, který je zodpovědný za výpočet působení gravitačních sil a následného pohybu těles.

2.5 Techniky optimalizace

2.5.1 Nastavení kompilátoru

V kompilátoru GCC je k dospozici mnoho různých nastavení, které řídí optimalizace kódu, které vznikají při překladu [8]. Můžeme použít jednotlivé možnosti postupně, nebo použít při překladu přepínač `-O`, případně `-O1`, `-O2`, `-O3`. Vyšší stupeň by měl znamenat rychlejší výsledný program, ale pomalejší překlad. V našem případě je velikost překládaného kódu relativně malá a není problém použít nejvyšší stupeň, tedy `-O3`.

Kompilátor se při tomto nastavení více či méně úspěšne pokusí o optimalizace jako je odstranění konstant, inline funkce, predikce skoku, antialiasing, vektorizace a další.

Kromě přepínačů, již zahrnutých v `-O3` nabízí kompilátor GCC i další, mimo jiné také na rozbalování cyklů. Tento je zde však nevyužit a technika rozbalovaní cyklů je zkoušena v kapitole 2.5.2 ne použitím tohoto přepínače, ale transformací zdrojového kódu a to z důvodu rozbalování vnořených cyklů.

2.5.1.1 FPU

FPU (Floating-point unit) je přídavný matematický koprocesor vytvořený speciálně pro počítání s čísly v plovoucí řádové čárce, který pracuje nezávisle na CPU [9]. Kromě instrukcí pro sčítání, odčítání, dělení a násobení podporuje také instrukci pro odmocňování, která bude využita, a další například pro goniometrické funkce, které v našem případě potřeba nebudou.

Pro využití FPU použijeme při kompilaci kromě volby úrovně optimalizací `-O3` také přepínač `-mhard-float`. Pro rozšíření sady koprocesoru o instrukci pro odmocňování pak použijeme další přepínač `-mfancy-math-387`.

2.5.1.2 Nastavení cílové architektury

Protože známe architekturu, ne které měřený program poběží, můžeme využít další možnost kompilátoru a přeložit kód optimálně pro naše CPU. Při kompilaci použijeme kromě volby úrovně optimalizací `-O3` přepínač `-march=ivybridge` a můžeme také přidat instrukční sady, které tento procesor podporuje přepínači `-msse4.2` a `-mavx`.

Při neznámé cílové architektuře můžeme použít `-march=generic`, obecné nastavení pro běžné procesory, nebo `-march=native` pro nastavení pro procesor použitý pro kompilaci [10].

2.5.2 Rozbalování cyklů

Rozbalení cyklu je optimalizační technika, která prodlouží vnitřek cyklu a tím umožní lepší plánování instrukcí uvnitř cyklu a sníží počet podmíněných skoků na (I/UF) + (I mod UF) + 1, kde I je počet iterací a UF znamená unrolling faktor, tedy kolikrát byl cyklus rozbalen [11]. Kompilátor GCC umožňuje volbu `-funroll-loop`, která „rozbalí cykly u kterých známe počet iterací při kompilaci nebo při vstupu do cyklu“ [8].

2.5.3 Způsob uložení dat v paměti

Až doposud bylo každé simulované těleso uloženo v paměti jako struktura s hodnotami jako jsou souřadnice, vektory rychlosti a hmotnost v jednom poli za sebou. Takovýto způsob uložení se nazývá pole struktur (array of structures, AOS).

Místo AOS můžeme pro naše data zvolit uložení, které se nazývá structure of arrays (SOA). Nyní tedy budeme mít několik samostatných polí pro hodnoty jednotlivých souřadnic, každou složku vektoru rychlosti i pro hmotnosti. Tento způsub uložení v paměti by měl zlepšit paměťovou efektivitu, pokud se nepracuje s celou původní strukturou, ale jen s některými položkami [11], nebo usnadnit vektorizaci.

2.5.4 vektorizace

Překladač GCC podporuje automatickou vektorizaci výpočtů. Pro její zapnutí je potřeba přepínač `-ftree-vectorize` a podpora vektorových sad, tedy přepínače `-mavx` nebo `-msse` [8]. Překladač se poté pokusí o vektorizaci smyček, které splňují následující podmínky:

- Vektorizuje se pouze nejvnitřnější cyklus.

- Vektorizovaný cyklus obsahuje pouze bezkolizní datové závislosti.
- Vektorizovaný cyklus provede dostatečný počet iterací.
- Vektorizovaný cyklus neobsahuje podmínky nebo skoky.
- Datové položky v cyklu jsou vhodně umístěny v paměti.

Poslední podmínka je splněna, pokus použijeme metodu uložení v paměti SOA z předchozí kapitoly. Podmínka na dostatečný počet iterací je zde proto aby se vektorizace vůbec vyplatila, nemá smysl vektorově počítat pouze například jen s jednou datovou položkou. Minimální počet iterací se dá nastavit přepínačem `min-vect-loop-bound`. Podle dokumentace je nicméně přednastavena hodnota 0.

2.5.5 Paralelizace

Vývoj procesorů se v posledních letech ubírá směrem k rozšiřování počtu výpočetních jader spíše než k navyšování výkonu jednoho jádra. Dnes v podstatě nenalezneme procesor, který by měl jen jedno jádro a i tak by díky hyperthreadingu zvládl počítat dvě vlákna programu zároveň. Systém na kterém měříme výkonnost simulace má k dospozici dva šestijádrové procesory, s hyperthreadingem tedy zvládne počítat až 24 vláken. Pokud se tedy podaří algoritmus vhodně paralelizovat, v ideálním případě se výpočet může zrychlit i 24krát.

Pro paralelizaci využijeme technologii OpenMP, což je multiplatformní API pro vícevláknové programování, jehož pomocí dokážeme vytvořit v kódu pomocí OpenMP direktiv takzvané paralelní bloky, v našem případě cykly, které se budou vykonávat vícevláknově [12].

Kapitola 3

Realizace

3.1 Popis použitého neoptimalizovaného algoritmu

Základní algoritmus pro N-Body simulaci má dvě hlavní části. Nejprve je nutné vypočítat síly působící na každé těleso v systému jako výslednice gravitačních působení mezi tělesy. Poté se výpočítají nové pozice a rychlosti těles vzniklé působením těchto sil po předem daný malý časový úsek. Opakováním těchto dvou kroků dosáhneme simulace pohybu těles v systému [3].

Takto popsaný algoritmus lze v jazyce C++ zapsat na několik desítek řádků.

```
1
2
3   for(int i = 1; i < bodies_count; i++)
4   {
5      ax = 0.0
6      ay = 0.0;
7      az = 0.0;
8
9      for (int j = 0; j < bodies_count; j++)
10       {
11        dx = bodies[j].x - bodies[i].x;
12        dy = bodies[j].y - bodies[i].y;
13        dz = bodies[j].z - bodies[i].z;
14        invrdist = 1.0/sqrt(dx*dx + dy*dy + dz*dz + EPS);
15        invrdist3 = invrdist*invrdist*invrdist;
16        f = -G*bodies[j].m * invrdist3;
17        ax += f*dx;
18        ay += f*dy;
19        az += f*dz;
20     }
21
22     tmp[i].x = bodies[i].x + dt*bodies[i].vx + 0.5*dt*dt*ax;
23     tmp[i].y = bodies[i].y + dt*bodies[i].vy + 0.5*dt*dt*ay;
24     tmp[i].z = bodies[i].z + dt*bodies[i].vz + 0.5*dt*dt*az;
```

```
25
26        bodies[i].vx -= dt*ax;
27        bodies[i].vy -= dt*ay;
28        bodies[i].vz -= dt*az;
29
30    }
31
32    for(int i = 0; i < bodies_count; i++)
33    {
34        bodies[i].x = tmp[i].x;
35        bodies[i].y = tmp[i].y;
36        bodies[i].z = tmp[i].z;
37    }
```

Řádek 3 je inicializace for cyklu přes všechna tělesa. Na řádcích 5-7 je inicializován vektor zrychlení pro dané těleso. Na řádku 9 pak začíná vnitřní cyklus přes všechna tělesa, kde se vypočítává vektor zrychlení tělesa „i“ podle gravitačního působení tělesa „j“ (řádky 16-19). K tomu potřebujeme vypočítat polohový vektor mezi tělesy (řádky 11-13). Dále pak potřebujeme třetí mocninu převrácené hodnoty vzdálenosti mezi tělesy - řádky 14 a 15.

Na řádcích 22-28 se poté aktualizují poloha a vektor rychlosti tělesa. Nová poloha tělesa je pak uložena do původního pole for cyklem (32-37). Za zmínku zde stojí fakt, že pokud bude $i == j$, počítá se vlastně gravitační vliv tělesa na sebe samé, což podle Newtonova gravitačního zákona není možné, taková síla by totiž byla nekonečná. Pokud však $i == j$, budou však hodnoty dx,dy a dz stejně jako výsledná síla rovny nule a tím se tento vliv tělesa na sebe vyloučí. Problém na řádku 14, kde by v takovém případě docházelo k dělení nulou, řeší malá konstanta *eps*, která této nesmyslné operaci zabrání. Problém, kdy $i == j$ by šel řešit také podmínkou, která by tuto rovnost na začátku každé iterace cyklu testovala. Vyhodnocení podmínky je ale časově náročná operace, kterou se nejspíš z tohoto důvodu nevyplatí v každé iteraci provádět. Přidání takové podmínky a testování jejího vlivu na výkonnost algoritmu bude jedním z předmětů v kapitole Optimalizace.

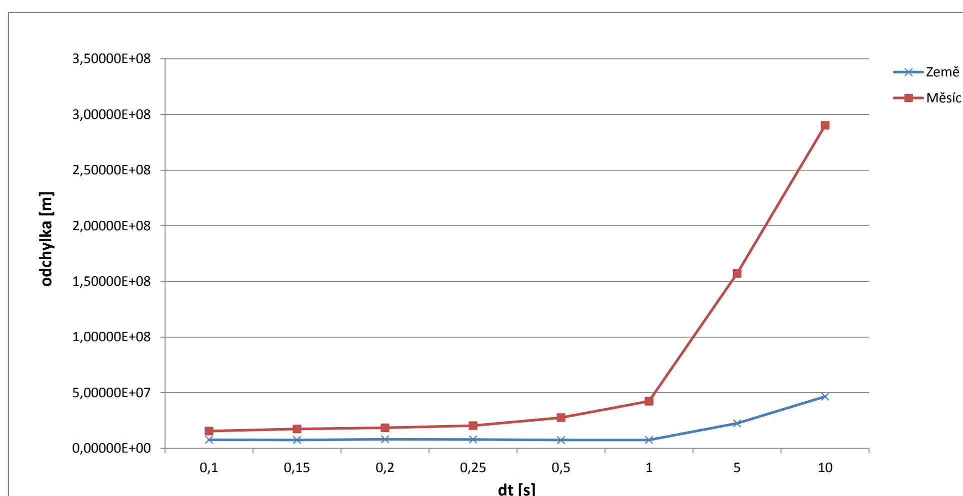
3.1.1 Volba vhodného časového kroku

Volba velikosti časového kroku jenoho cyklu je důležitou součástí simulace, ovlivňující nejen přesnost ale i rychlost celé simulace. Předpokládejme, že čím menší krok zvolíme, tím přesnější simulace bude. Vzhledem k fyzikálním zákonům je tento předpoklad jistě správný, když si ale uvědomíme, jak jsou uožena desetinná čísla v paměti počítače, tento předpoklad již nemusí být zcela správný z důvodu zaokrouhlovacích chyb ke kterým zde dochází.

Pro vyzkoušení různých časových kroků *dt*, zvolíme simulaci pohybu trojice Slunce, Země, Měsíc. Vliv ostatních těles prozatím zanedbáme. Tabulka 3.1 a graf 3.1 ukazují, jaký vliv má volba *dt* na přesnost simulace pohybu těchto tří těles po 500 dnech (43200000 sekundách). V tabulce jsou odchylky

dt[s]	odchylka Země [m]	odchylka Měsíce [m]
0,1	7,71733E+06	1,55166E+07
0,15	7,57705E+06	1,74177E+07
0,2	8,13455E+06	1,84869E+07
0,25	8,01408E+06	2,04225E+07
0,5	7,41330E+06	2,76325E+07
1	7,58333E+06	4,23358E+07
5	2,24701E+07	1,57252E+08
10	4,65024E+07	2,90303E+08

Tabulka 3.1: Odchylky pro různé časové kroky



Obrázek 3.1: Odchylky pro různé časové kroky

poloh Měsíce a Země od skutečné polohy podle [5] pro různé časové kroky *dt*.

Z grafu 3.1 lze vyčíst, že pokud zvolíme krok větší než 1s, odchylka začne výrazně narůstat. Zajímavé je, že při kroku 0,5 sekundy je odchylka Země, narozdíl od Měsíce, menší než při kroku 0,25 sekundy. Nicméně krok velikosti 0,25 sekundy vypadá jako vhodný kompromis mezi přesností simulace a její rychlostí.

3.2 Použití programu a jeho funkcionalita

Program je napsán v jazyce C++, pro jeho překlad je tedy potřeba překladač GCC a to ve verzi nejlépe 4.9.3 a vyšší. Zdrojové kódy, které jsou na přiloženém CD ve složce src, lze přeložit pomocí tamtéž přiloženého makefilu. Lze přeložit program v různých verzích, pro různé typy a možnosti optimalizace.

Příkaz `make` tak přeloží základní verzi pouze s vhodnými nastaveními kompilátoru podle kapitoly 2.5.1.
`make WOConst` - verze s předpočítanou konstantou z kapitoly 4.2.2.2.
`make Cond` - verze s přidanou podmínkou z kapitoly 4.2.2.1.
`make Unroll2` - verze s dvakrát rozbaleným cyklem z kapitoly 2.5.2.
`make Unroll4` - čtyřikrát rozbalený.
`make Unroll42` - čtyřikrát plus dvakrát rozbalený.
`make SOA` - verze se způsobem uložení dat SOA z kapitoly 2.5.3.
`make Vector` - vektorizovaná verze, kapitola 2.5.4
`make Parallel4` - Paralelní verze pro 4 vlákna. Kapitola 2.5.5
`make Parallel6` - Paralelní verze pro 6 vláken.
`make Parallel8` - Paralelní verze pro 8 vláken.
`make Parallel12` - Paralelní verze pro 12 vláken.
`make Parallel16` - Paralelní verze pro 16 vláken.
`make Parallel24` - Paralelní verze pro 24 vláken.

Program se spustí z konzole a musí mít dva argumenty. První argument je vstupní soubor s daty o tělesech sluneční soustavy, přiložený ukázkový vstupní soubor System20150101.txt představuje data pro několik největších těles sluneční soustavy v čase 2015-01-01 00:00:00. Druhý argument je délka simulace v sekundách. Výstupem programu je pak stav soustavy po provedení příslušně dlouhé simulace vypsaný na standartní výstup ve formátu stejném jako jsou vstupní data. Takto uložený výstup pak tedy může být vstupním souborem další simulace.

Pro vizualizaci výstupu simulace je zapotřebí nástroje Gnuplot. Zadáním souboru s daty soustavy ve formátu stejném jako je soubor System20150101.txt přiloženému skriptu DisplaySystem.gp se zobrazí 3D graf představující tento zadaný stav soustavy.

3.2.1 Přidávání vlastních těles

Přidat vlastní tělesa, lze například úpravou přiloženého souboru System20150101.txt, nebo i vytvořením jiného, vlastního a to v následujícím formátu. Každý řádek představuje jedno těleso a ve sloupcích jsou následující údaje v tomto pořadí: název (označení) tělesa, souřadnice polohy x, souřadnice polohy y, souřadnice polohy z, složka vektoru rychlosti x, složka vektoru rychlosti y, složka rychlosti z, hmotnost tělesa. Souřadnice jsou vzhledem ke Slunci, které představuje bod (0,0,0). Na prvním řádku pak musí zůstat Slunce se souřadnicemi 0,0,0. Takové údaje lze získat například z webových stránek NASA [5].

Kapitola 4

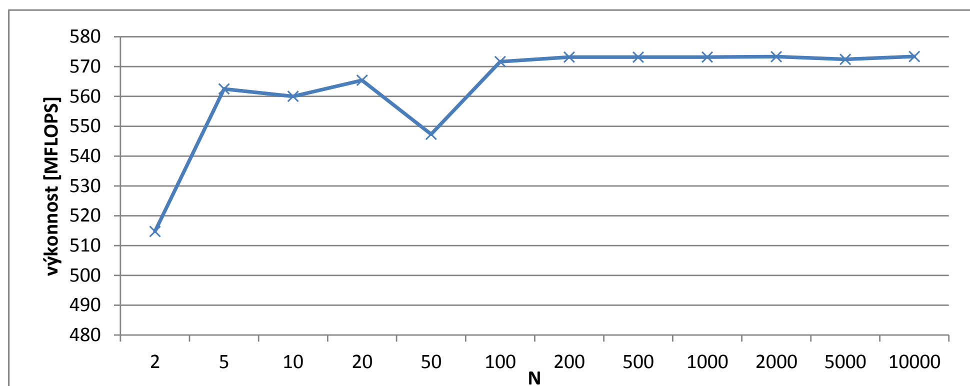
Měření výkonnosti

Následující kapitoly obsahují měření výkonnosti různých verzí zdrojového kódu pro různé objemy vstupních dat. Veškerá měření byla prováděna na výpočetním klastru STAR (star.fit.cvut.cz), kde byly k dospozici dva šestijádrové procesory Intel Xeon E5-2620 v2 2,1 GHz s 6 x 256 KB L2 Cache a 15 MB sdílené L3 Cache [13] [14].

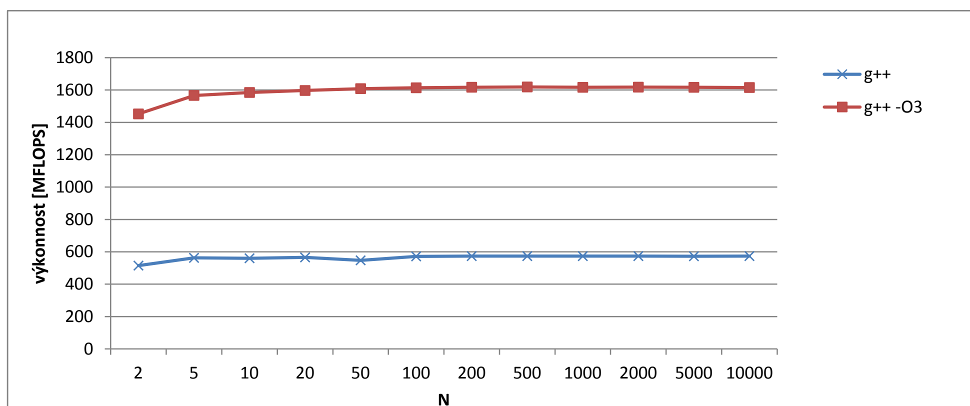
4.1 Výkonnost neoptimalizovaného algoritmu

Výkonnost neoptimalizovaného algoritmu takového, jak je popsán v kapitole 3.1, kompilovaného kompilátorem GCC ve verzi 4.9.3 bez jakýchkoli nastavených optimalizací. Výsledky měření je možno vidět v tabulce C.1 a na grafu 4.1. N představuje množství simulovaných objektů, I (počet cyklů) znamená kolik bylo provedeno iterací (posunů) soustavy. Počet cyklů je volen tak, aby byl v každém měření vykonán přibližne stejný počet FP instrukcí, tento počet je anyliticky vyjádřen z charakteristik 2 vnořených for cyklů jako $(18 * počettěles + 22 * (počettěles + 1) * počettěles) * početcyklů$, ve vnějším cyklu je totiž 18 FP operací a ve vnitřním 22. Výpočetní čas pak v sekundách udává, jak dlouho simulace trvala. A konečně poslední sloupec představuje výkonnost v milionech operací v plovoucí řádové čárce za sekundu.

Z výsledků měření je možné vypozorovat, že objem vstupních dat nemá negativní vliv na výkonnost, jak se očekávalo. Důvodem může být, že velikost dat není dostatečná na zaplnění 15 MB L3 cache procesoru. Malá výkonnost pro malý počet těles lze přičíst nezanedbatelné režii cyklů, kterých bylo pro malý počet těles velmi mnoho.



Obrázek 4.1: Výkonnost neoptimalizovaného algoritmu



Obrázek 4.2: Výkonnost neoptimalizovaného algoritmu

4.2 Optimalizované verze

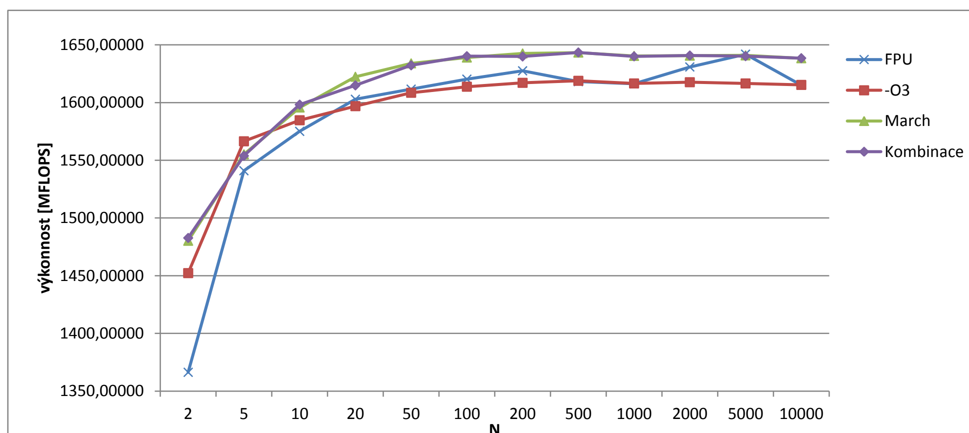
Tato kapitola obsahuje měření výkonnosti optimalizovaných algoritmů získaných různými nastaveními kompilátoru, metodami transformací zdrojového kódu a jeho paralelizací.

4.2.1 Nastavení kompilátoru

Výsledky měření výkonnosti algoritmu se zapnutými optimalizacemi při překladu jsou v tabulce C.2, srovnání se stejným kódem bez zapnutých optimalizací pak na grafu 4.2

Výkonnost algoritmu s použitím FPU je možno vidět v tabulce C.3. Srovnání s ostatními nastaveními kompilátoru pak na grafu 4.3.

Výkonnost programu optimalizovaného na cílovou architekturu je v tabulce C.4. Srovnání s ostatními nastaveními kompilátoru pak na grafu 4.3.



Obrázek 4.3: Výkonnost pro různé nastavení kompilátoru

4.2.1.1 Kombinace různých nastavení kompilátoru

Jelikož při zkoušení použití FPU i optimalizace pro cílovou architekturu jsme pro většinu objemů vstupních dat dosáhli zlepšení, zkusíme tyto zkombinovat a změřit, zda dosáhneme ještě lepšího výsledku.

Výsledky měření takto zkombinovaných nastavení při kompilaci jsou v tabulce C.5. Srovnání s ostatními nastaveními kompilátoru pak na grafu 4.3.

Přidání použití FPU v kombinaci s nastavením pro cílovou architekturu nepřináší žádné zlapšení, ale spíše naopak. FPU tak nebude v dalších testech využíváno.

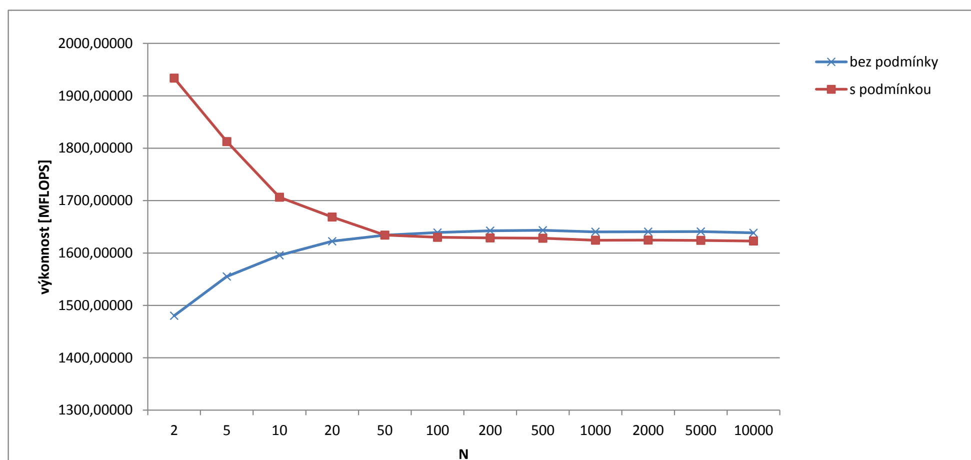
4.2.2 Drobné úpravy kódu

4.2.2.1 Přidání podmínky

Přidání podmínky do vnitřního cyklu, která tento přeskočí, pokud by mělo jít o působení tělesa samo na sebe, tak jak je vidět na následující části zdrojového kódu.

```
1           .
2           .
3           .
4           for (int j = 0; j < bodies_count; j++)
5           {
6           if (i==j) continue;
7           .
8           .
9           .
```

Tato podmínka by měla způsobit zrychlení zejména v případě malého počtu těles v simulaci, kdy bude k této rovnosti docházet často, nicméně vyhodnocení podmínky je poměrně náročná operace a pro větší počty těles může



Obrázek 4.4: Srovnání výkonnosti kódu s podmínkou a bez

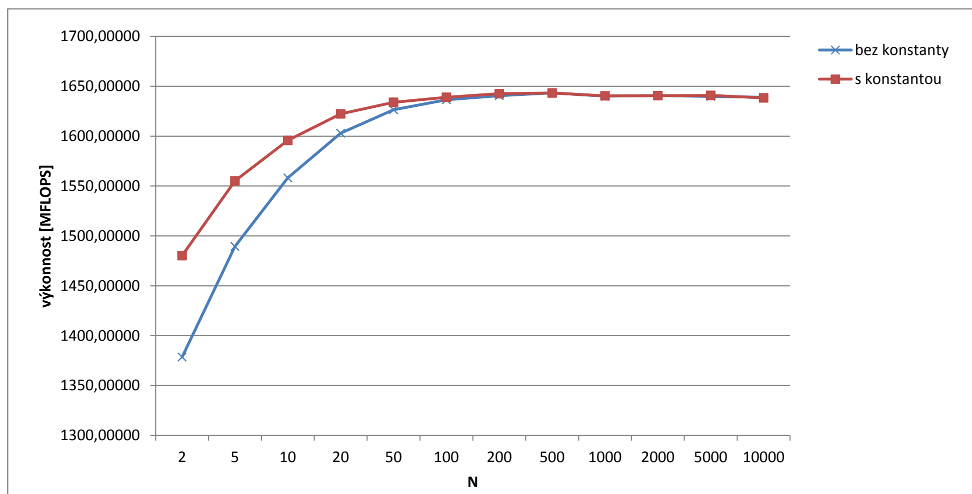
naopak představovat zbytečné prodloužení výpočetního času. Výsledky měření jsou v tabulce C.6. Vliv podmínky na snížení počtu vykonaných FP operací z důvodu srovnání s předchozím kódem zanedbáme.

Na grafu 4.4 je pak vidět srovnání s výkonností předchozího kódu bez podmínky ve vnitřním cyklu. Podle předpokladu se výpočet pro malý počet těles významně zrychlil. Takto upravený kód pak přestává být výhodou pro simulaci více než padesáti těles. S přibývajícím počtem simulovaných objektů pak výkonnost programu s podmínkou zaostává za tím bez podmínky ne víc než o jedno procento. Toto snížení výkonnosti je zanedbatelné s ohledem na to, že pokud budeme chtít například simulovat pouze planety a několik dalších větších těles ve sluneční soustavě nepřekročíme toto „kritické“ množtví padesáti těles.

4.2.2.2 Odstranění konstanty

Konstanta $0.5 * dt * dt$ se v následující trojici řádků počítá zbytečne nejen třikrát, ale také znovu v každém cyklu smyčky „i“.

```
1          for(int i = 1; i < bodies_count; i++){
2          .
3          .
4          .
5          tmp[i].x = bodies[i].x + dt*bodies[i].vx + 0.5*dt*dt*ax;
6          tmp[i].y = bodies[i].y + dt*bodies[i].vy + 0.5*dt*dt*ay;
7          tmp[i].z = bodies[i].z + dt*bodies[i].vz + 0.5*dt*dt*az;
8          .
9          .
10         .
11         }
```



Obrázek 4.5: Výkonnost pro různé nastavení kompilátoru

Nahradíme ji proto jedním operandem vypočítaným jen jednou ještě před cyklem.

```
double ConstDT2 = 0.5*dt*dt;

for(int i = 1; i < bodies_count; i++){
.
.
.
tmp[i].x = bodies[i].x + dt*bodies[i].vx + ConstDT2*ax;
tmp[i].y = bodies[i].y + dt*bodies[i].vy + ConstDT2*ay;
tmp[i].z = bodies[i].z + dt*bodies[i].vz + ConstDT2*az;
.
.
.
}
```

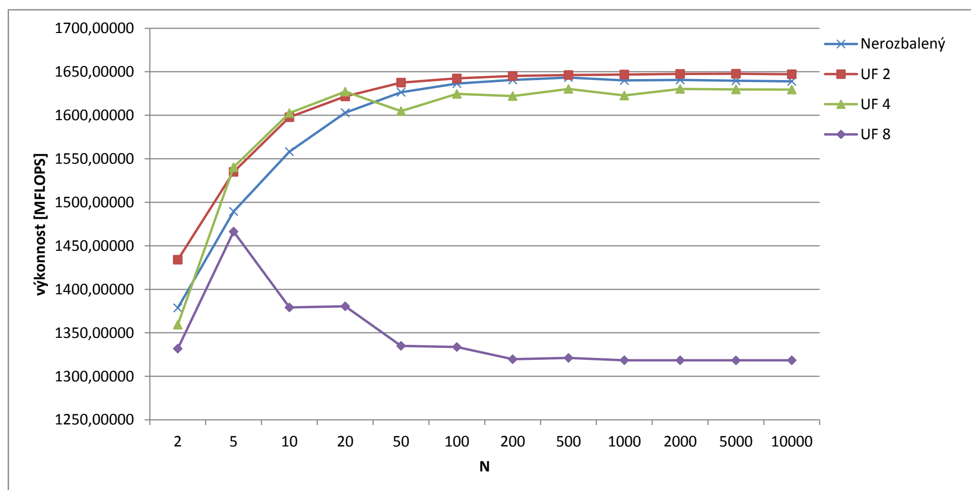
Výsledky měření takto upraveného kódu jsou v tabulce C.7. Srovnání s původním kódem pak na obrázku 4.5.

Pokud započítáme i snížení počtu instrukcí, kterého dosáhneme tím že konstantu předpočítáme, zjistíme, že výkonnost spíše klesla, nicméně výpočetní časy zůstali stejné, nebo alespoň velmi podobné. O tuto práci s předpočítáním konstanty, se tedy spolehlivě postará už kompilátor přesně podle předpokladu a toho co se píše v jeho dokumentaci [8].

4.2.3 Rozbalování cyklů

Tabulku výkonnosti různě rozbaleného kódu můžeme vidět v tabulkách C.8, C.9, C.10, srovnání s nerozbaleným kódem pak na grafu na obrázku 4.6.

Rozbalování v průměru nejlépe dopadlo pro faktor 2. Pro faktor 4 nemá



Obrázek 4.6: Srovnání nerozbaleného kódu s rozbalováním pro různé faktory rozbalení

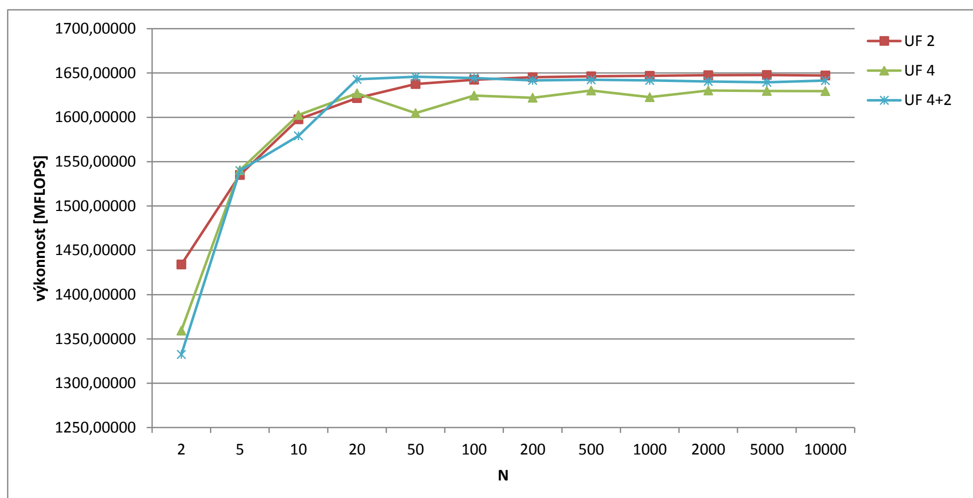
rozbalení pro 2 tělesa žádný smysl a vykoná se stejný kód jako v nerozbaleném případě, pro větší objem vstupních dat (5-20) pak začíná být čtyřikrát rozbalený kód efektivnější, později však zaostává, nevýhody větší spotřeby registrů a více výpadků v cache začínají převažovat nad výhodami snížené režie cyklu. U osmkrát rozbaleného kódu pak tento problém nastává mnohem dříve a projeví se s mnohem větší intenzitou.

U čtyřikrát rozbaleného cyklu pak ještě můžeme zkusit přidat kód rozbalený s faktorem 2, který se bude vykonávat, pokud po čtyřikrát rozbaleném cyklu zbydou 3 nebo 2 iterace. Naměřené výsledky výkonnosti jsou v tabulce C.11 a srovnání s ostatními faktory rozbalení pak na obrázku 4.7.

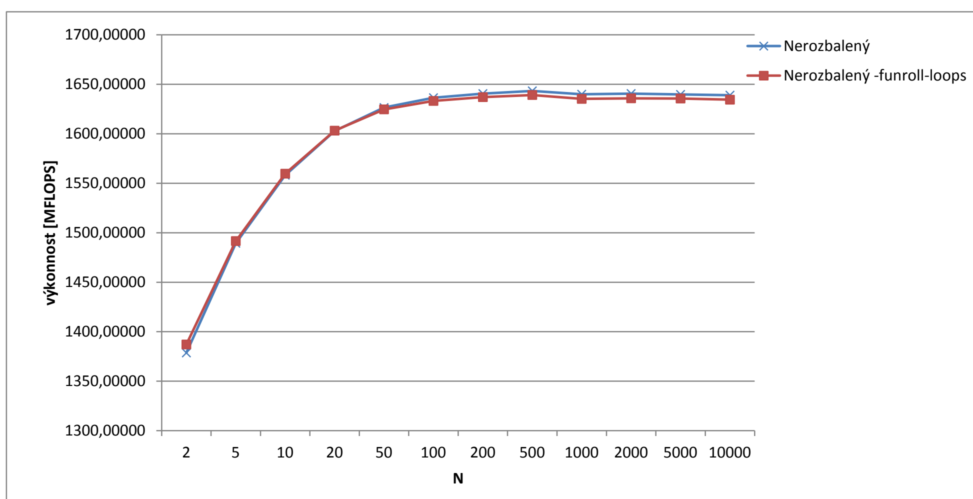
Vzhledem k voleným velikostem dat se však tato změna projeví jen pro $N = 2$, $N = 10$, a $N = 50$, a lepšího výsledku jsme dosáhli jen pro $N = 50$. Tato úprava tedy pravděpodobně nemá smysl a zůstaneme u $UF = 2$, který se zdá být tím nejoptimálnějším.

4.2.3.1 Volba -funroll-loops

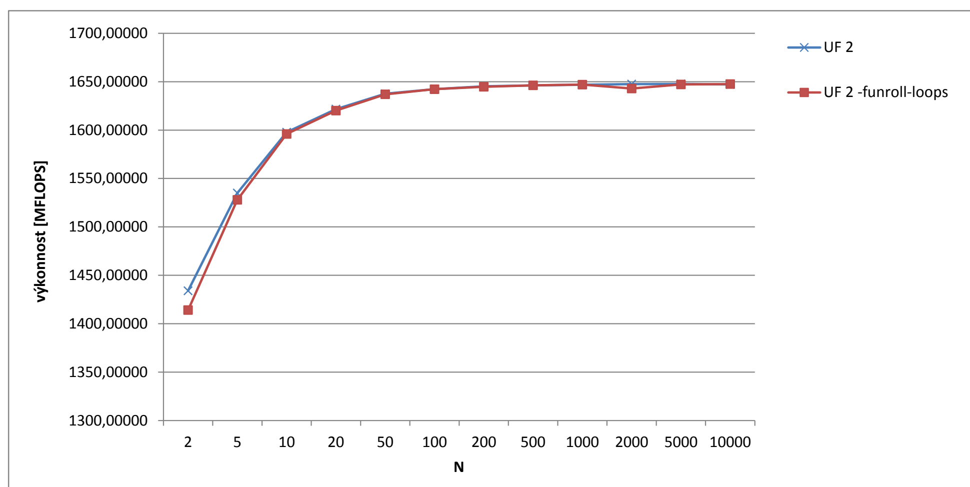
Teď, když máme vyzkoušeno, jak výkonnost ovlivní rozbalení vnějšího cyklu ve zdrojovém kódu, můžeme tuto změnu srovnat s tím, jak výkonnost ovlivní volba `-funroll-loops` při kompilaci. Tuto volbu použijeme jak na nerozbalený kód, tak i na rozbalený kód s UF 2 a 4+2, kde by mělo dojít k rozbalení minimálně vnitřního cyklu. Naměřenou výkonnost můžeme vidět v tabulkách C.12, C.13 a C.14, srovnání s jejich verzemi bez volby `-funroll-loops` pak na grafech na obrázcích 4.8, 4.9 respektive 4.10.



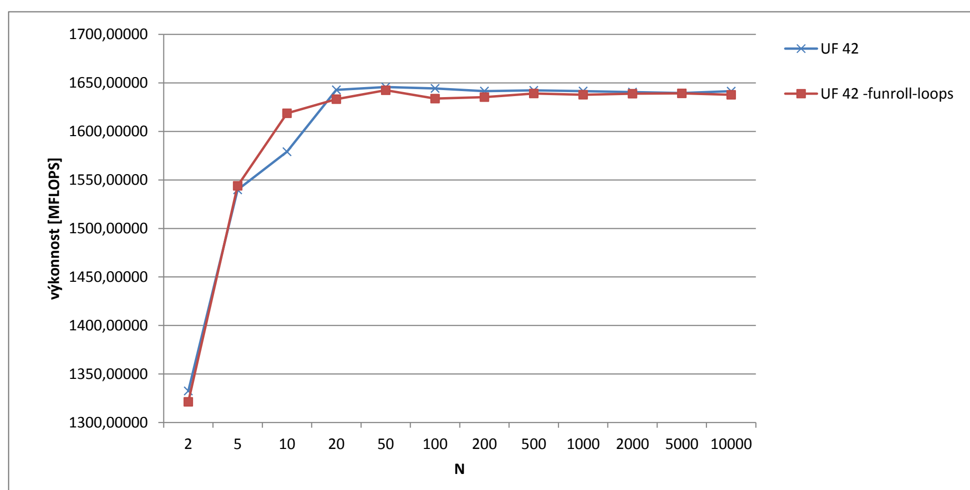
Obrázek 4.7: Srovnání kódu s kombinovaným faktorem rozbalení



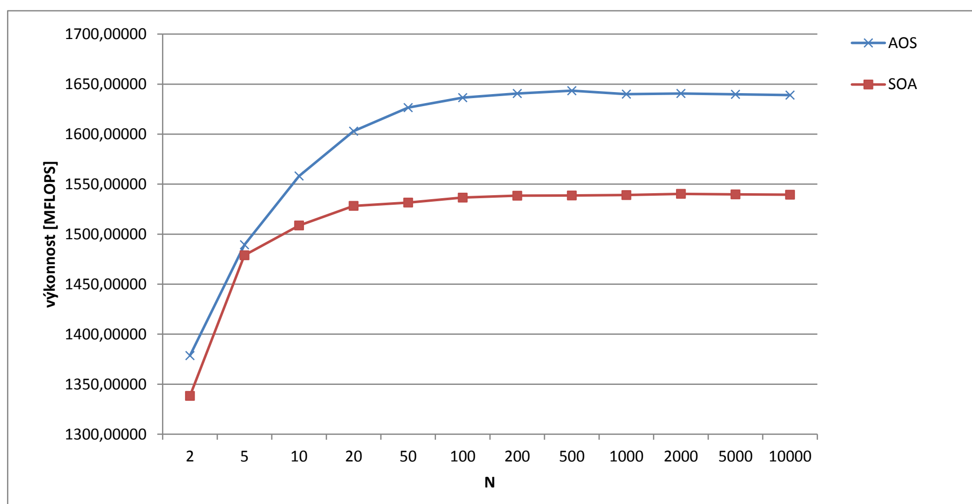
Obrázek 4.8: Srovnání nerozbaleného kódu bez a s volbou -funroll-loops



Obrázek 4.9: Srovnání dvakrát rozbaleného kódu bez a s volbou -funroll-loops



Obrázek 4.10: Srovnání čtyřikrát plus dvakrát rozbaleného kódu bez a s volbou -funroll-loops



Obrázek 4.11: Srovnání způsobů uložení dat v paměti metodami AOS a SOA

Ve většině případů nedosahujeme při použití přepínače -funroll-loops při kompilaci žádného zlepšení a jeho použití tak nemá pro tento kód smysl.

4.2.4 Způsob uložení dat v paměti

Naměřenou výkonnost s novým způsobem uložení dat je možno vidět v tabulce C.15, srovnání výkonnosti kódu s uložením dat AOS z kapitoly 4.2.2.2 pak na grafu 4.11.

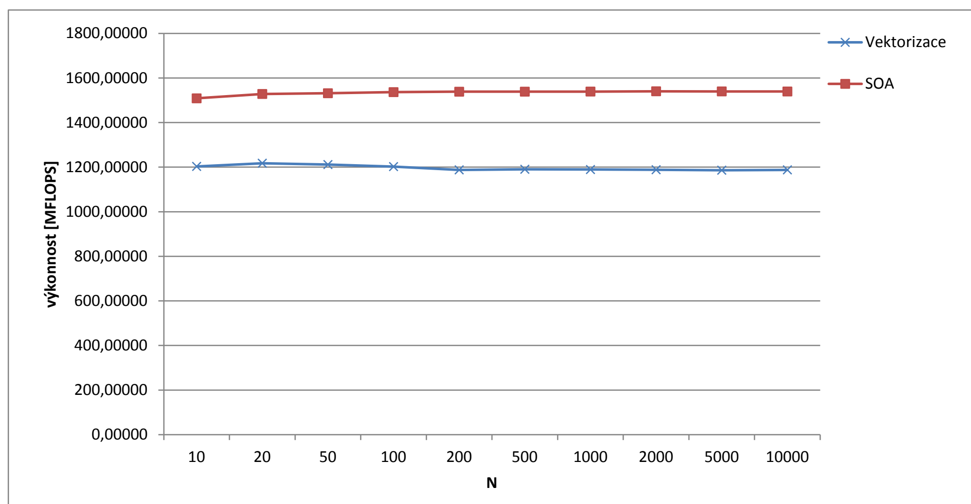
Samotná změna uložení dat jako SOA nám tedy nepřinesla zlepšení, ale právě naopak. To ale neznamená, že je špatná a nevyužijeme ji v kombinaci s jinou metodou optimalizace. Vyzkoušejme tedy SOA v kombinaci s rozbalováním cyklů z kapitoly 4.2.3. V tabulce C.17 jsou výsledky měření uložení SOA s dvakrát rozbaleným vnějším cyklem.

Pro dvakrát rozbalený cyklus jsme pro AOS i SOA dostali téměř totožné výsledky. Ani v tomto případě tedy není SOA výhodou, ale potenciál SOA ještě využijeme v kapitolách 4.2.5 Vektorizace a 4.2.6 Paralelizace.

4.2.5 vektorizace

Úpravou kódu z předchozí kapitoly tak, aby byla co největší jeho část ve vektorizovatelných cyklech dosáhneme výkonnosti, kterou lze vidět v tabulce C.16 a srovnání s nevektorizovanou verzí pak na grafu 4.12. Vektorizace nebyla provedena pro malá vstupní data, protože by se nevyplatila.

Vektorizace nepřinesla žádné zrychlení, ale právě naopak. Důvodem je, že vektorizována byla pouze malá část výpočtu a to z důvodu datových kolizí, kvůli kterým není možno provést vektorizaci na větší části programu.



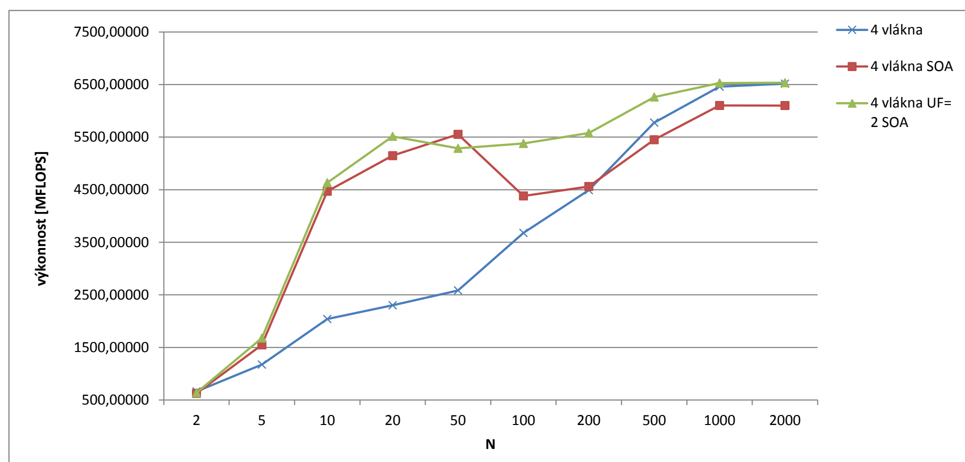
Obrázek 4.12: Srovnání vektorizace s nevektorizovanou verzí

4.2.6 Paralelizace

Jako první musíme stanovit, které části kódu budeme paralelizovat. Chceme dosáhnout paralelizace v co největší části kódu, ale vzhledem k datovým závislostem mezi jednotlivými iteracemi („posuny soustavy“) nemůžeme tedy paralelizovat celý výpočet, ale jen v rámci jedné iterace. Nabízí se tedy paralelizace vnějšího cyklu, cyklu který představuje jedno těleso, pro které se poté počítají síly, kterými na něj působí ostatní tělesa. Paralelizací tohoto cyklu tedy fakticky dosáhneme toho, že se bude počítat pohyb více těles najednou, jeden pohyb tělesa vždy proběhne v rámci jednoho vlákna. Z toho tedy vyplývá, že nemá smysl paralelizovat pro malé počty těles, kde režie vláken bude větší než čas ušetřený paralelním výpočtem. Začneme proto měřit výkonnost až pro soustavu deseti těles, ale jak později měření ukáží, zrychlení dosáhneme až od počtu dvaceti těles a více. Také nemá smysl používat více vláken než máme v soustavě těles.

Kód upravíme na příslušných místech následujícími direktivami `pragma omp parallel num_threads(T)` na začátku simulace vytvoří T vláken a direktiva `pragma omp for schedule(static)` paralelizuje náš zvolený for cyklus. Zvolili jsme nastavení plánovače (schedule) jako static, to znamená rovnoměrně mezi vlákna. Pro jiné rozdělení není důvod, protože předpokládáme rovnoměrné rozložení práce v jednotlivých iteracích cyklu a tak jiné plánování nepřináší žádné výhody. Static by také mělo mít nejmenší nároky na režii.

Zbývá vyřešit otázka, který z verzí kódu z předchozích kapitol, bude nejvhodnější k paralelizaci. Teoreticky by to mohl být kód ještě před rozbalováním cyklů, pro kód dvakrát rozbalený budeme totiž moci fakticky použít pro malé



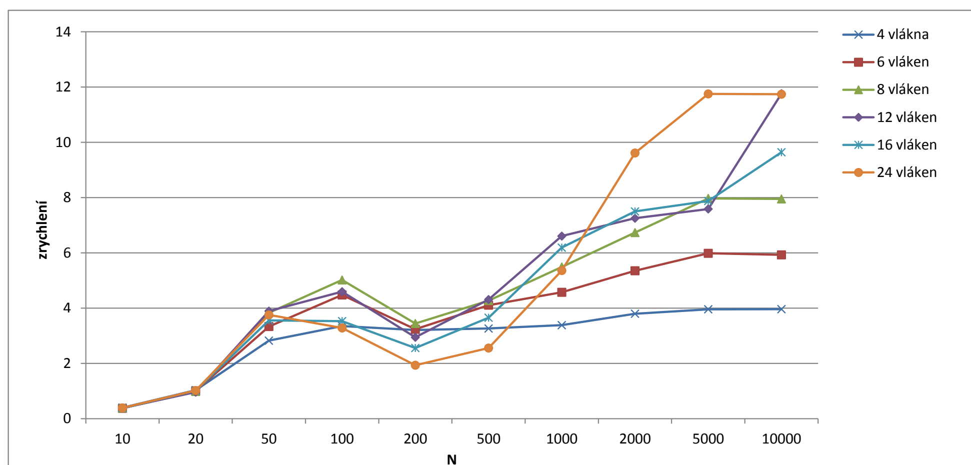
Obrázek 4.13: Srovnání různých verzí kódu paralelizovaných 4 vlákny

počty těles N pouze $N/2$ vláken. Naměřenou výkonnost pro paralelní verze se 4 vlákny nerozbaleného kódu, kódu s metodou uložení dat SOA a kombinaci SOA s rozbalováním cyklů můžeme vidět v tabulkách C.18, C.19 respektive C.20, srovnání těchto verzí pak na grafu 4.13.

Srovnání paralelních verzí dopadlo nejlépe pro kombinaci SOA a rozbalení cyklů. Výrazného zrychlení, z důvodu režie a čekání vláken a paměťové efektivity, dosáhneme i v tomto případě až pro počty těles v řádu stovek, pro deset výrazně zaostává za sekvenční verzí, pro dvacet je srovnatelná, v případě padesáti těles je zrychlení asi trojnásobné. Očekávané čtyřnásobné zrychlení dosáhneme až pro počty těles v řádu tisíců.

Výkonnost nejlepší paralelní verze pro více vláken (6, 8, 12, 16 a 24) jsou v následujících tabulkách C.21, C.22, C.23, C.24 a C.25, srování jejich zrychlení oproti sekvenční verzi pak na obrázku 4.14, pro menší počty těles je ale potřeba si uvědomit, že můžeme využít pouze omezený počet vláken a sice pro 10 maximálně 5 vláken a pro 20 těles pak 10 vláken.

Výhody paralelizace se výrazněji projeví až při větších počtech těles N. Do počtu dvanácti vláken dokážeme dosáhnout pro N v řádu tisíců i lineárního zrychlení, pro více než dvanáct vláken je pak cache efektivita nižší a výkonnost v některých případech klesá.



Obrázek 4.14: Srovnání zrychlení pro různé počty vláken

Závěr

Podařilo se implementovat algoritmus, který více či méně přesně simuluje pohyb planet v systému. Tento algoritmus se poté podařilo v průběhu práce pomocí různých optimalizačních technik a vhodnou paralelizací zrychlit, v závislosti na velikosti vstupních dat, až 35 krát, což lze brát jako úspěch. Použité optimalizace pak mohou být zdrojem inspirace i pro jiné, podobné, úlohy.

Práce se zabývá pouze optimalizací pro CPU, tento problém by však mohl být vhodný i pro paralelní výpočty na grafických kartách, například pomocí technologie CUDA. Také by šlo vylepšit vizualizaci pohybu planet například nějakou vhodnou animací.

Literatura

[1] Pretorius, F.: N-body Simulations. Online [cit. 2015-10-2]. Dostupné z: http://physics.princeton.edu/~fpretori/Nbody/intro.htm

[2] Wikipedia, the free encyclopedia - Newton's law of universal gravitation. Online [cit. 2015-10-5]. Dostupné z: https://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation

[3] Brown Deer Technology, L.: Basic N-Body Algorithm. 2009-2010, online [cit. 2015-10-7]. Dostupné z: http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html

[4] Narlikar, G.: Parallel N-Body Simulations. Online [cit. 2015-10-18]. Dostupné z: http://www.cs.cmu.edu/~scandal/alg/nbody.html

[5] of Technology, J. P. L. C. I.: HORIZONS Web-Interface. Online [cit. 2015-11-02]. Dostupné z: http://ssd.jpl.nasa.gov/horizons.cgi

[6] Havrlant, R.: Simulátor sluneční soustavy. 2008. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/havrlr1_2008bach.pdf

[7] Havrlant, R.: Pokročilý simulátor sluneční soustavy. 2010. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/havrlrad_2010dipl.pdf

[8] Free Software Foundation, I.: Options That Control Optimization. Online [cit. 2015-10-2]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[9] Šimeček, I.; Šoch, M.: Matematický koprocesor v procesorech x86. České vysoké učení technické v Praze, Katedra počítačových systémů FIT, přednáška, 2014.

[10] Šimeček, I.: Pokročilá nastavení kompilátoru GCC. České vysoké učení technické v Praze, Katedra počítačových systémů FIT, přednáška, 2014.

[11] Šimeček, I.; Šoch, M.: Kompilátorové optimalizace I: Metody transformací zdrojových kódů. České vysoké učení technické v Praze, Katedra počítačových systémů FIT, přednáška, 2014.

[12] Šimeček, I.: Technologie OpenMP. České vysoké učení technické v Praze, Katedra počítačových systémů FIT, přednáška, 2014.

[13] Wikipedia, the free encyclopedia - List of Intel Xeon microprocessors. Online [cit. 2015-10-15]. Dostupné z: https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors

[14] Intel®: Intel® Xeon® Processor E5-2620 v2 (15M Cache, 2.10 GHz). Online [cit. 2015-10-15]. Dostupné z: http://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz

Příloha A

Seznam použitých zkratek

GCC GNU Compiler Collection

FP Floating point

FPU Floating point unit

CPU Central processing unit

UF Unrolling factor

AOS Array of structures

SOA Structure of arrays

NASA National Aeronautics and Space Administration

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
exe ........................ adresář se spustitelnou formou implementace
src
  impl ........................ zdrojové kódy implementace
  thesis ........................ zdrojová forma práce ve formátu LaTeX
text ........................ text práce
  thesis.pdf ........................ text práce ve formátu PDF
```

Příloha C

Tabulky naměřených výkonností

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	1554,200	514,73427
5	1025641026	8,00000E+11	1422,150	562,52857
10	300751880	8,00000E+11	1428,470	560,03976
20	82304527	8,00000E+11	1414,830	565,43896
50	13961606	8,00000E+11	1461,690	547,31169
100	3561888	8,00000E+11	1399,380	571,68178
200	899686	8,00001E+11	1395,700	573,18965
500	144849	8,00001E+11	1395,760	573,16518
1000	36288	8,00005E+11	1395,700	573,19284
2000	9082	8,00052E+11	1395,460	573,32460
5000	1454	8,00034E+11	1397,620	572,42628
10000	364	8,00967E+11	1396,920	573,38104

Tabulka C.1: Výkonnost neoptimalizovaného algoritmu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	550,877	1452,22981
5	1025641026	8,00000E+11	510,714	1566,43444
10	300751880	8,00000E+11	504,836	1584,67304
20	82304527	8,00000E+11	500,979	1596,87333
50	13961606	8,00000E+11	497,365	1608,47672
100	3561888	8,00000E+11	495,746	1613,72970
200	899686	8,00001E+11	494,716	1617,09100
500	144849	8,00001E+11	494,181	1618,84214
1000	36288	8,00005E+11	494,879	1616,56738
2000	9082	8,00052E+11	494,588	1617,61212
5000	1454	8,00034E+11	494,897	1616,56753
10000	364	8,00967E+11	495,862	1615,30313

Tabulka C.2: Výkonnost neoptimalizovaného algoritmu se zapnutými optimalizacemi při překladu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	585,518	1366,31154
5	1025641026	8,00000E+11	519,161	1540,94780
10	300751880	8,00000E+11	507,918	1575,05739
20	82304527	8,00000E+11	499,109	1602,85629
50	13961606	8,00000E+11	496,398	1611,61009
100	3561888	8,00000E+11	493,745	1620,26966
200	899686	8,00001E+11	491,559	1627,47664
500	144849	8,00001E+11	494,397	1618,13487
1000	36288	8,00005E+11	494,970	1616,27017
2000	9082	8,00052E+11	490,577	1630,83786
5000	1454	8,00034E+11	487,254	1641,92479
10000	364	8,00967E+11	495,927	1615,09141

Tabulka C.3: Výkonnost algoritmu s použitím FPU

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	540,455	1480,23425
5	1025641026	8,00000E+11	514,423	1555,14042
10	300751880	8,00000E+11	501,353	1595,68209
20	82304527	8,00000E+11	493,110	1622,35607
50	13961606	8,00000E+11	489,613	1633,94359
100	3561888	8,00000E+11	488,107	1638,98499
200	899686	8,00001E+11	487,052	1642,53671
500	144849	8,00001E+11	486,829	1643,28959
1000	36288	8,00005E+11	487,679	1640,43407
2000	9082	8,00052E+11	487,655	1640,60974
5000	1454	8,00034E+11	487,572	1640,85390
10000	364	8,00967E+11	488,880	1638,37228

Tabulka C.4: Výkonnost algoritmu s nastavením cílové architektury při kompilaci

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	539,553	1482,70883
5	1025641026	8,00000E+11	514,856	1553,83253
10	300751880	8,00000E+11	500,557	1598,21958
20	82304527	8,00000E+11	495,401	1614,85343
50	13961606	8,00000E+11	490,123	1632,24338
100	3561888	8,00000E+11	487,750	1640,18461
200	899686	8,00001E+11	487,838	1639,89027
500	144849	8,00001E+11	486,790	1643,42124
1000	36288	8,00005E+11	487,802	1640,02043
2000	9082	8,00052E+11	487,608	1640,76788
5000	1454	8,00034E+11	487,772	1640,18111
10000	364	8,00967E+11	488,858	1638,44601

Tabulka C.5: Výkonnost algoritmu s kombinací nastavení kompilátoru

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	8,00000E+11	413,692	1933,80583
5	1025641026	8,00000E+11	441,347	1812,63269
10	300751880	8,00000E+11	468,805	1706,46644
20	82304527	8,00000E+11	479,424	1668,66907
50	13961606	8,00000E+11	489,557	1634,13050
100	3561888	8,00000E+11	490,826	1629,90560
200	899686	8,00001E+11	491,161	1628,79543
500	144849	8,00001E+11	491,360	1628,13625
1000	36288	8,00005E+11	492,499	1624,37944
2000	9082	8,00052E+11	492,445	1624,65157
5000	1454	8,00034E+11	492,602	1624,09901
10000	364	8,00967E+11	493,542	1622,89621

Tabulka C.6: Výkonnost algoritmu s přidanou podmínkou na rovnost těles

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	541,606	1378,61594
5	1025641026	7,69231E+11	516,459	1489,43240
10	300751880	7,81955E+11	501,838	1558,18190
20	82304527	7,90123E+11	492,923	1602,93486
50	13961606	7,95812E+11	489,280	1626,49514
100	3561888	7,97863E+11	487,552	1636,46731
200	899686	7,98921E+11	486,962	1640,62323
500	144849	7,99566E+11	486,534	1643,39282
1000	36288	7,99788E+11	487,677	1639,99434
2000	9082	7,99943E+11	487,571	1640,66887
5000	1454	7,99991E+11	487,863	1639,78576
10000	364	8,00946E+11	488,672	1639,02495

Tabulka C.7: Výkonnost algoritmu s předpočítanou konstantou

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	520,684	1434,01116
5	1025641026	7,69231E+11	501,136	1534,97408
10	300751880	7,81955E+11	489,398	1597,78930
20	82304527	7,90123E+11	487,226	1621,67754
50	13961606	7,95812E+11	485,978	1637,54644
100	3561888	7,97863E+11	485,796	1642,38263
200	899686	7,98921E+11	485,606	1645,20448
500	144849	7,99566E+11	485,679	1646,28588
1000	36288	7,99788E+11	485,646	1646,85289
2000	9082	7,99943E+11	485,539	1647,53513
5000	1454	7,99991E+11	485,506	1647,74647
10000	364	8,00946E+11	486,244	1647,20922

Tabulka C.8: Výkonnost dvakrát rozbaleného kódu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	549,274	1359,37013
5	1025641026	7,69231E+11	499,397	1540,31916
10	300751880	7,81955E+11	487,951	1602,52748
20	82304527	7,90123E+11	485,597	1627,11767
50	13961606	7,95812E+11	495,893	1604,80495
100	3561888	7,97863E+11	491,137	1624,52210
200	899686	7,98921E+11	492,529	1622,07945
500	144849	7,99566E+11	490,453	1630,26117
1000	36288	7,99788E+11	492,871	1622,71166
2000	9082	7,99943E+11	490,681	1630,27009
5000	1454	7,99991E+11	490,878	1629,71410
10000	364	8,00946E+11	491,511	1629,55783

Tabulka C.9: Výkonnost čtyřikrát rozbaleného kódu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	560,635	1331,82314
5	1025641026	7,69231E+11	524,605	1466,30469
10	300751880	7,81955E+11	566,977	1379,16509
20	82304527	7,90123E+11	572,357	1380,47313
50	13961606	7,95812E+11	596,103	1335,02355
100	3561888	7,97863E+11	598,231	1333,70372
200	899686	7,98921E+11	605,409	1319,63874
500	144849	7,99566E+11	605,202	1321,15637
1000	36288	7,99788E+11	606,584	1318,51074
2000	9082	7,99943E+11	606,734	1318,44030
5000	1454	7,99991E+11	606,809	1318,35685
10000	364	8,00946E+11	607,501	1318,42680

Tabulka C.10: Výkonnost osmkrát rozbaleného kódu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	560,355	1332,48863
5	1025641026	7,69231E+11	499,562	1539,81041
10	300751880	7,81955E+11	495,194	1579,08797
20	82304527	7,90123E+11	480,947	1642,84934
50	13961606	7,95812E+11	483,552	1645,76207
100	3561888	7,97863E+11	485,209	1644,36956
200	899686	7,98921E+11	486,673	1641,59748
500	144849	7,99566E+11	486,851	1642,32276
1000	36288	7,99788E+11	487,201	1641,59663
2000	9082	7,99943E+11	487,622	1640,49727
5000	1454	7,99991E+11	487,935	1639,54379
10000	364	8,00946E+11	487,945	1641,46697

Tabulka C.11: Výkonnost čtyřikrát plus dvakrát rozbaleného kódu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	538,297	1387,09052
5	1025641026	7,69231E+11	515,698	1491,63031
10	300751880	7,81955E+11	501,345	1559,71414
20	82304527	7,90123E+11	492,843	1603,19505
50	13961606	7,95812E+11	489,818	1624,70865
100	3561888	7,97863E+11	488,519	1633,22801
200	899686	7,98921E+11	488,021	1637,06309
500	144849	7,99566E+11	487,772	1639,22177
1000	36288	7,99788E+11	489,081	1635,28642
2000	9082	7,99943E+11	489,000	1635,87436
5000	1454	7,99991E+11	489,093	1635,66193
10000	364	8,00946E+11	490,033	1634,47278

Tabulka C.12: Výkonnost kódu s volbou -funroll-loops

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	528,024	1414,07714
5	1025641026	7,69231E+11	503,437	1527,95835
10	300751880	7,81955E+11	489,948	1595,99567
20	82304527	7,90123E+11	487,688	1620,14128
50	13961606	7,95812E+11	486,141	1636,99738
100	3561888	7,97863E+11	485,840	1642,23389
200	899686	7,98921E+11	485,744	1644,73708
500	144849	7,99566E+11	485,692	1646,24182
1000	36288	7,99788E+11	485,611	1646,97159
2000	9082	7,99943E+11	486,893	1642,95350
5000	1454	7,99991E+11	485,679	1647,15954
10000	364	8,00946E+11	486,140	1647,56161

Tabulka C.13: Výkonnost dvakrát rozbaleného kódu s volbou -funroll-loops

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	565,101	1321,29773
5	1025641026	7,69231E+11	498,228	1543,93324
10	300751880	7,81955E+11	483,073	1618,70957
20	82304527	7,90123E+11	483,771	1633,25925
50	13961606	7,95812E+11	484,529	1642,44357
100	3561888	7,97863E+11	488,339	1633,83001
200	899686	7,98921E+11	488,530	1635,35744
500	144849	7,99566E+11	487,805	1639,11087
1000	36288	7,99788E+11	488,324	1637,82145
2000	9082	7,99943E+11	488,099	1638,89408
5000	1454	7,99991E+11	488,031	1639,22128
10000	364	8,00946E+11	489,054	1637,74471

Tabulka C.14: Výkonnost čtyřikrát plus dvakrát rozbaleného kódu s volbou -funroll-loops

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	557,904	1338,34256
5	1025641026	7,69231E+11	520,130	1478,92021
10	300751880	7,81955E+11	518,292	1508,71495
20	82304527	7,90123E+11	516,998	1528,29113
50	13961606	7,95812E+11	519,598	1531,59085
100	3561888	7,97863E+11	519,256	1536,55020
200	899686	7,98921E+11	519,301	1538,45490
500	144849	7,99566E+11	519,637	1538,70198
1000	36288	7,99788E+11	519,651	1539,08589
2000	9082	7,99943E+11	519,334	1540,32388
5000	1454	7,99991E+11	519,547	1539,78524
10000	364	8,00946E+11	520,277	1539,45994

Tabulka C.15: Výkonnost kódu s uložením dat metodou SOA

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	649,824	1203,33335
20	82304527	7,90123E+11	648,879	1217,67457
50	13961606	7,95812E+11	656,830	1211,59439
100	3561888	7,97863E+11	663,531	1202,45009
200	899686	7,98921E+11	672,607	1187,79788
500	144849	7,99566E+11	671,600	1190,53973
1000	36288	7,99788E+11	672,505	1189,26628
2000	9082	7,99943E+11	673,112	1188,42416
5000	1454	7,99991E+11	674,338	1186,33504
10000	364	8,00946E+11	674,313	1187,79499

Tabulka C.16: Výkonnost vektorizovaného algoritmu

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
2	4444444444	7,46667E+11	520,142	1435,50543
5	1025641026	7,69231E+11	506,893	1517,54072
10	300751880	7,81955E+11	487,503	1604,00016
20	82304527	7,90123E+11	485,863	1626,22686
50	13961606	7,95812E+11	484,997	1640,85869
100	3561888	7,97863E+11	485,166	1644,51530
200	899686	7,98921E+11	485,060	1647,05638
500	144849	7,99566E+11	485,014	1648,54309
1000	36288	7,99788E+11	485,190	1648,40067
2000	9082	7,99943E+11	485,153	1648,84595
5000	1454	7,99991E+11	485,067	1649,23773
10000	364	8,00946E+11	485,760	1648,85046

Tabulka C.17: Výkonnost dvakrát rozbaleného kódu s uložením dat metodou SOA

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1180,360	662,47152
20	82304527	7,90123E+11	672,891	1174,22206
50	13961606	7,95812E+11	390,014	2040,46917
100	3561888	7,97863E+11	346,604	2301,94375
200	899686	7,98921E+11	309,161	2584,15896
500	144849	7,99566E+11	217,376	3678,26476
1000	36288	7,99788E+11	178,049	4491,95177
2000	9082	7,99943E+11	138,488	5776,25903
5000	1454	7,99991E+11	123,835	6460,13486
10000	364	8,00946E+11	122,916	6516,20294

Tabulka C.18: Výkonnost paralelizovaného kódu pro čtyři vlákna

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1249,160	625,98457
20	82304527	7,90123E+11	510,274	1548,42978
50	13961606	7,95812E+11	178,037	4469,92222
100	3561888	7,97863E+11	155,021	5146,80535
200	899686	7,98921E+11	143,884	5552,53654
500	144849	7,99566E+11	182,493	4381,35424
1000	36288	7,99788E+11	175,365	4560,70208
2000	9082	7,99943E+11	146,783	5449,83111
5000	1454	7,99991E+11	131,113	6101,53684
10000	364	8,00946E+11	131,290	6100,58344

Tabulka C.19: Výkonnost paralelizovaného kódu s uložením dat metodou SOA pro čtyři vlákna

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1225,530	638,05446
20	82304527	7,90123E+11	471,050	1677,36643
50	13961606	7,95812E+11	171,788	4632,52114
100	3561888	7,97863E+11	144,689	5514,33013
200	899686	7,98921E+11	151,148	5285,68799
500	144849	7,99566E+11	148,659	5378,52723
1000	36288	7,99788E+11	143,403	5577,20215
2000	9082	7,99943E+11	127,743	6262,12442
5000	1454	7,99991E+11	122,529	6528,99150
10000	364	8,00946E+11	122,577	6534,22420

Tabulka C.20: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro čtyři vlákna

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1309,000	597,36813
20	82304527	7,90123E+11	482,866	1636,32034
50	13961606	7,95812E+11	145,462	5470,92397
100	3561888	7,97863E+11	108,392	7360,90221
200	899686	7,98921E+11	149,943	5328,16582
500	144849	7,99566E+11	118,113	6769,50446
1000	36288	7,99788E+11	106,103	7537,84078
2000	9082	7,99943E+11	90,647	8824,81009
5000	1454	7,99991E+11	81,061	9868,99742
10000	364	8,00946E+11	81,947	9773,94658

Tabulka C.21: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro šest vláken

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1262,770	619,23778
20	82304527	7,90123E+11	495,227	1595,47735
50	13961606	7,95812E+11	126,574	6287,32237
100	3561888	7,97863E+11	96,673	8253,21353
200	899686	7,98921E+11	140,849	5672,18204
500	144849	7,99566E+11	113,493	7045,07309
1000	36288	7,99788E+11	88,387	9048,70083
2000	9082	7,99943E+11	72,048	11102,91139
5000	1454	7,99991E+11	60,849	13147,14786
10000	364	8,00946E+11	61,097	13109,40963

Tabulka C.22: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro osm vláken

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1279,280	611,24608
20	82304527	7,90123E+11	506,697	1559,36084
50	13961606	7,95812E+11	124,582	6387,85332
100	3561888	7,97863E+11	105,581	7556,87967
200	899686	7,98921E+11	164,845	4846,49924
500	144849	7,99566E+11	112,518	7106,12062
1000	36288	7,99788E+11	73,445	10889,61155
2000	9082	7,99943E+11	66,899	11957,46663
5000	1454	7,99991E+11	63,928	12513,93443
10000	364	8,00946E+11	41,298	19394,29512

Tabulka C.23: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro dvanáct vláken

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1271,200	615,13128
20	82304527	7,90123E+11	480,169	1645,51118
50	13961606	7,95812E+11	136,357	5836,23534
100	3561888	7,97863E+11	137,475	5803,69458
200	899686	7,98921E+11	189,684	4211,85323
500	144849	7,99566E+11	132,746	6023,28115
1000	36288	7,99788E+11	78,376	10204,49525
2000	9082	7,99943E+11	64,697	12364,29307
5000	1454	7,99991E+11	61,587	12989,60495
10000	364	8,00946E+11	50,383	15897,13991

Tabulka C.24: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro šestnáct vláken

N	I	počet FP instrukcí	výpočetní čas [s]	výkonnost [MFLOPS]
10	300751880	7,81955E+11	1252,070	624,52969
20	82304527	7,90123E+11	477,415	1655,00342
50	13961606	7,95812E+11	129,138	6162,48929
100	3561888	7,97863E+11	147,891	5394,93892
200	899686	7,98921E+11	250,874	3184,55148
500	144849	7,99566E+11	189,872	4211,08157
1000	36288	7,99788E+11	90,528	8834,69777
2000	9082	7,99943E+11	50,461	15852,68940
5000	1454	7,99991E+11	41,272	19383,37856
10000	364	8,00946E+11	41,374	19358,66970

Tabulka C.25: Výkonnost dvakrát rozbaleného paralelizovaného kódu s uložením dat metodou SOA pro dvacet čtyři vláken