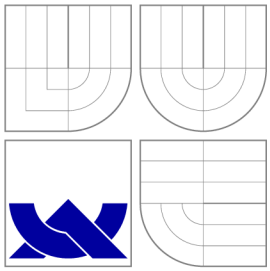
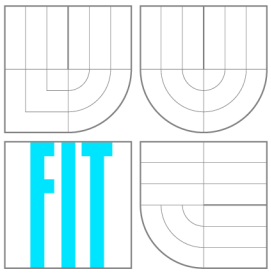


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VYSOCE NÁROČNÉ APLIKACE NA SVAZKU KARET INTEL XEON PHI

HIGH PERFORMANCE APPLICATIONS ON INTEL XEON PHI CLUSTER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ KAČURIK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2016

Abstrakt

Táto práca sa zaoberá implementáciou a optimalizáciou vysoko náročných aplikácií na zväzku Intel Xeon Phi koprocesorov. Na dvoch prístupoch k riešeniu N-Body problému boli demonštrované možnosti behu programov na zväzku procesorov, koprocesorov a s využitím oboch typov zariadení. Zvolené boli dva verzie N-Body problému - naivná a Barnes-hut. Oba problémy boli implementované a optimalizované. Práca tiež zachytáva proces optimalizácie a zmeny vo výkone po aplikovaní jednotlivých optimalizácií. Pre lepšie porovnanie dosiahnutých výkonov sme porovnávali programy na základe dosiahnutého zrýchlenia voči behu programu na jednom výpočtovom uzle pri využití len procesorov. V prípade naivnej verzie bolo dosiahnuté 15 násobné zrýchlenie pri využití procesorov a koprocesorov na 8 výpočtových uzloch. Výkon dosiahnutý v tomto prípade predstavoval 9 TFLOP/s. Na základe dosiahnutých výsledkov sme v závere zhodnotili výhody a nevýhody pri behu programov v distribuovanom prostredí na procesoroch, koprocesoroch alebo s využitím oboch typov zariadení.

Abstract

The main topic of this thesis is the implementation and subsequent optimization of high performance applications on a cluster of Intel Xeon Phi coprocessors. Using two approaches to solve the N-Body problem, the possibilities of the program execution on a cluster of processors, coprocessors or both device types have been demonstrated. Two particular versions of the N-Body problem have been chosen - the naive and Barnes-hut. Both problems have been implemented and optimized. For better comparison of the achieved results, we only considered achieved acceleration against single node runs using processors only. In the case of the naive version a 15-fold increase has been achieved when using combination of processors and coprocessors on 8 computational nodes. The performance in this case was 9 TFLOP/s. Based on the obtained results we concluded the advantages and disadvantages of the program execution in the distributed environments using processors, coprocessors or both.

Kľúčové slová

Intel Xeon Phi, MIC, Salomon klaster, Optimalizácia, Vektorizácia, N-Body, Barnes-Hut, OpenMP, MPI, Natívny mód, Hybridný mód

Keywords

Intel Xeon Phi, MIC, Salomon cluster, Optimalizations, Vectorization, N-Body, Barnes-Hut, OpenMP, MPI, Native mode, Hybrid mode

Citácia

KAČURIK, Tomáš. *Vysoce náročné aplikace na svazku karet Intel Xeon Phi.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Jaroš Jiří.

Vysoce náročné aplikace na svazku karet Intel Xeon Phi

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Jiřího Jaroša, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Kačurik
25. mája 2016

Poďakovanie

Chcel by som poďakovať vedúcemu mojej diplomovej práce Ing. Jiřímu Jarošovi PhD. za poskytnutú pomoc a odborné rady. Táto práca bola podporovaná Ministerstvom školstva, mládeže a telovýchovy z projektu "IT4Innovations National Supercomputing Center – LM2015070" pre podporu veľkých výskumných infraštruktúr, experimentálny vývoj a inovatívne projekty.

© Tomáš Kačurik, 2016.
Táto práca vznikla ako školské dielo na FIT VUT v Brně. Práca je chránená autorským zákonom a jej využitie bez poskytnutia oprávnenia autorom je nezákonné, s výnimkou zákonne definovaných prípadov.

Obsah

Kapitola 1

Úvod

V súčasnej dobe je trendom zvyšovať počet jadier procesorov, namiesto zvyšovania frekvencie. Zvyšovanie frekvencie narazilo na strop fyzikálnych možností súčasných technológií. Jedným z riešení je použitie viacero jadier s nižšou frekvenciou. Intel po rokoch výskumu prišiel na trh s koprocesorom Intel Xeon Phi, ako odpoveď na rýchlo rozvíjajúcu sa sféru GPGPU systémov. Tieto koprocesory začínajú prenikať do oblasti vysoko náročných výpočtov v rámci superpočítačov. Oproti existujúcim riešeniam založeným na GPGPU, Intel Xeon Phi karty ponúkajú známe prostredie x86 architektúry.

Cieľom tejto práce je priblížiť čitateľovi problémy spojené s implementáciou, optimalizáciou a behom programov na zväzku Intel Xeon Phi koprocesorov. Porovnaný bude výkon pri behu implementovaných riešení N-Body problému v niekoľkých konfiguráciách. V závere práce, po odmeraní a porovnaní výsledkov behu programov, budú prezentované výhody a nevýhody zvolených konfigurácií.

V úvode práce je čitateľ oboznámený s teóriou z danej oblasti. Predstavený bude stručný prehľad hardvérovej architektúry Intel Xeon Phi koprocesora a jeho dôležitých komponentov. Taktiež čitateľ bude oboznámený s výpočtovým klastrom Salomon, v rámci ktorého bola táto práca vytvorená. Tento klaster obsahuj 432 výpočtových uzlov s Intel Xeon Phi koprocesormi, čo predstavuje spolu 846 koprocesorov. Štvrtá kapitola poskytuje prehľad niektorých optimalizačných techník, ktoré je možné využiť pri optimalizácií algoritmov pre Intel Xeon platformu (procesory a koprocesory) v distribuovanom prostredí. Teoretickým popisom N-Body problému a aproximáciou pomocou Barnes-Hut algoritmu sa zaoberá piata kapitola.

V praktickej časti sa práca zaoberá implementáciou a optimalizáciou zvolených N-Body problémov. Optimalizácie týchto problémov následne čiastočne vychádzajú z nameraných výsledkov získaných z výkonnostných testoch klastra Salomon. Proces implementácie a optimalizácie je popísaný v siedmej kapitole. Táto kapitola tiež obsahuje merania dopadu jednotlivých optimalizácií na beh programov.

V závere práce sa zaoberáme porovnávaním výkonu implementovaných N-Body simulácií. Tieto programy sú použité na demonštráciu možností výpočtového klastra. Zhodnotené sú behy programov na procesoroch, koprocesoroch a v kombinovanom, tzv. hybridnom móde. Na základe tohto zhodnotenia a dosiahnutých výsledkov sú prezentované výhody a nevýhody jednotlivých architektúr výpočtových jednotiek.

Kapitola 2

Intel Xeon Phi

V roku 2012 Intel vydal prvú generáciu koprocesorov, ktorá vychádzala z výskumných programov "80-core Tera-Scale Computing Research Programä "Single-Chip Cloud Computer initiative". Ich cieľom bolo preskúmať nové možnosti paralelizmu s využitím väčšieho množstva výpočtových jadier a navrhnúť architektúru s vysokým výpočtovým výkonom a zároveň energetický úspornú. Výsledkom tejto iniciatívy bola nová architektúra, Intel MIC (angl. Intel Many Core) a rada produktov na nej založených - Intel Xeon Phi.

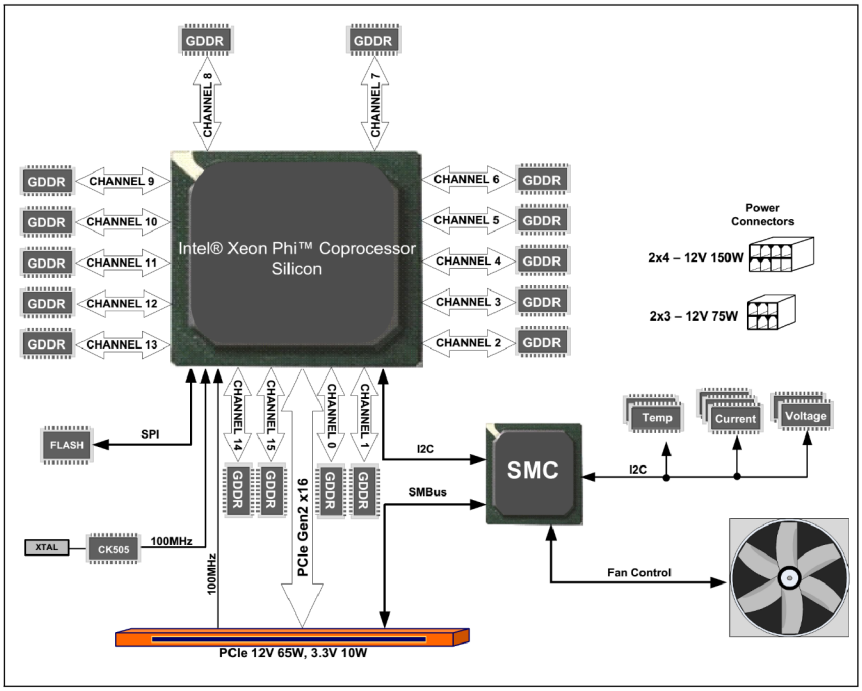
Táto kapitola je založená na [9].

2.1 Základné informácie

Intel Xeon Phi koprocesory sú PCI Express rozširujúce karty založené na Intel Many Integrated Core (Intel MIC) architektúre. Tieto koprocesory obsahujú až 61 procesorových jadier, ktoré sú založené na x86 architektúre. Na Intel Xeon Phi koprocesor sa môžeme pozerať ako na základnú dosku, ktorá obsahuje všetky potrebné komponenty pre beh programov, ktorá je pripojená k hostiteľskému počítaču pomocou PCI Express zbernice. Na každej karte pripojenej k hostiteľskému počítaču beží samostatný operačný systém, založený na Linuxe. Pre tento operačný systém je vyhradené 1 jadro. Každý pripojený a správne nakonfigurovaný Intel Xeon Phi koprocesor má pridelenú ip adresu a je možné sa naň prihlásiť pomocou SSH protokolu. Priamo v operačnom systéme bežiacom na karte je možné spúšťať programy, ktoré bežne nájdeme v Linuxových distribúciách ako vi, perl alebo awk.

Intel Xeon Phi pozostáva z výpočtových jadier, cache pamätí (angl. cache memory), pamäťových radičov (angl. memory controller), GDDR5 pamäťových čipov a ďalších podporných komponentov. Schematický pohľad na hlavné komponenty koprocesora je zobrazený na obrázku 2.1. Výpočtové jadrá, cache pamäte a pamäťové radiče sú umiestnené v "Ball Grid Array"puzdre.

Každé jadro koprocesora má priradenú 512 KB L2 cache pamäť. Naviac, každé jadro má prístup do L2 cache pamätí ostatných jadier cez ODI (angl. on-die interconnect). Použitím distribuovaného "Tag Directory"mechanizmu, prístupy do cache pamäte sú koherentné (dáta v cache sú konzistentné naprieč všetkými jadrami) bez softvérového zásahu. Zjednodušene sa na koprocesor môžeme pozerať ako na symetrický multiprocesor (SMP) so zdieľaným uniformným prístupom do pamäte (UMA). Každé jadro má rovnakú prioritu nezávisle na fyzickom umiestnení pamäte.



Obr. 2.1: Schéma hlavných komponentov Intel Xeon Phi koprocesora

2.1.1 Architektúra výpočtových jadier

Každé jadro Intel Xeon Phi koprocesora bolo navrhnuté s ohľadom na spotrebu pri zachovaní čo najvyššej priepustnosti vo vysoko paralelných úlohách. Obrázok[1] 2.2 zobrazuje schému architektúry Výpočtového jadra koprocesora. Architektúra vychádza z Intel Pentium P54c architektúry (in-order superscalar). Táto architektúra bola vylepšená, bola pridaná podpora 64 bitovej inštrukčnej sady, podpora vektorových inštrukcií, jadro dokáže vykonávať 4 hardvérové vlákna a ďalšie. Ďalej dokáže jadro spracovať 2 inštrukcie v 1 cykle, jednu na "V-Pipe"("Pipe 0"na obrázku 2.2) a jednu na Ü-Pipe"("Pipe 1"na obrázku 2.2). "V-pipe"sa používa predovšetkým na vykonávanie vektorových inštrukcií.

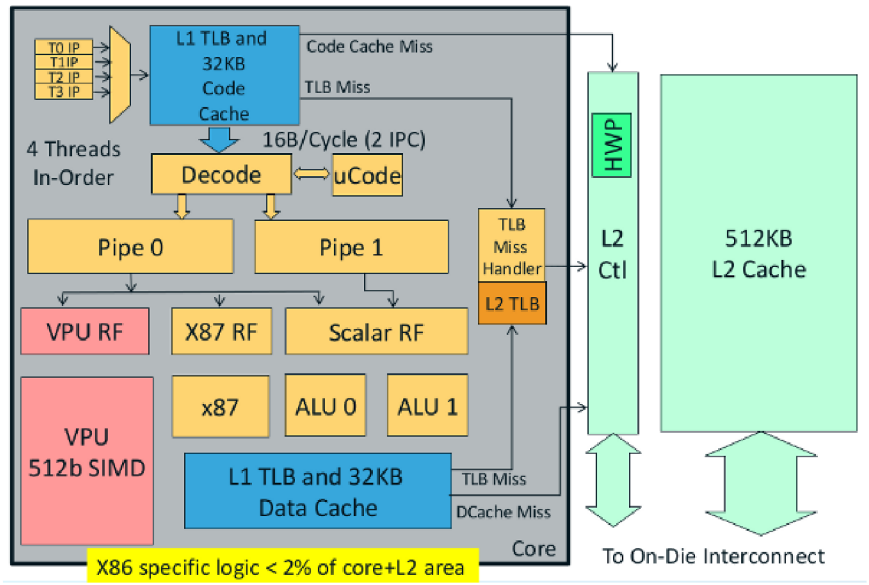
Dekodér inštrukcií je navrhnutý tak, že spracuje inštrukciu každé 2 cykly, čo ale spôsobuje, že využitie jadra pri behu 1 vlákna na jadro je 50%. Avšak, pri využití viacerých hardvérových vlákien, inštrukcia môže byť naplánovaná na každý cyklus. Väčšina skalárnych inštrukcií má latenciu 1 hodinový tik, zatiaľ čo väčšina vektorových inštrukcií má latenciu 4 hodinové tiky a priepustnosť 1 hodinový tik.

Niektoré registre sú replikované štyri krát, pre každé hardvérové vlákno. Každé hardvérové má osobitné aritmetické a segmentové registre, CR, DR EFLAGS a EIP. Taktiež sú replikované prefetch buffre, inštrukčný pointer (angl. instruction pointer), segmentové deskriptory a logika výnimiek. Intel Xeon Phi koprocesor implementuje round-robin multithreading.

2.1.2 Organizácia cache pamäte

Okrem vyššie spomenutej 512 KB L2 cache pamäte, jednotlivé jadrá obsahujú dve L1 cache pamäte - cache inštrukcií a cache dát. Dáta (a inštrukcie) v L1 sa zároveň nachádzajú aj v L2

[1] http://www.theregister.co.uk/2012/09/05/intel_xeon_phi_coprocessor/



Obr. 2.2: Schéma jadra Intel Xeon Phi koprocesora

cache pamäti. L2 sú plne koherentné v rámci celého systému a môžu medzi sebou navzájom prenášať dáta. Dáta zdieľané viacerými jadrami sú umiestnené v každej L2 príslušného jadra. Teda celkové množstvo efektívne využiteľnej L2 cache je funkcia miery zdieľania dát medzi jadrami (512 KB až $N * 512KB$, kde N je počet jadier).

Veľkosť L1 pamätí je 32 KB (32 KB pre dáta + 32 KB pre inštrukcie). Cache pamäte v Intel Xeon Phi koprocesore sú 8-cestné (angl. 8-way) asociatívne s veľkosťou "cache line"64 bajtov. Prístupová doba do L1 je 1 cyklus, do L2 11 cyklov.

2.1.3 Vector processing unit

Vector processing unit, ďalej len VPU, je komponent ktorý sa stará o vykonávanie vektorových, respektíve SIMD inštrukcií. Intel Xeon Phi nepodporuje staršie SIMD inštrukčné sady ako MMX alebo SSEx, namiesto poskytuje novú inštrukčnú sadu Intel Initial Many Core Instructions. Inštrukcie z tejto sady pracujú nad 512 bitovými vektormi. Podporované sú aj FMA (Fused Multiply-Add) inštrukcie. Ide o inštrukcie, ktoré v sebe spájajú 2 operácie - násobenie a sčítanie (respektíve odčítanie). VPU dokáže spracovať 16 single-precision(SP) alebo 8 double-precision(DP) elementov v jednom cykle. K dispozícií je 32 512 bitových registrov a 8 16 bitových maskovacích registrov. Nachádza sa tu aj Extended Math Unit (EMU), ktorá podporuje niektoré matematické operácie (exponent, logaritmus a ďalšie). Podporované sú IEEE 754 2008 desatinné čísla.

2.1.4 Direct Memory Access

Priamy prístup do pamäte (angl. Direct memory access), ďalej len DMA, je funkcia hardvéru, ktorá umožňuje priamy prenos dát medzi operačnou pamäťou a ďalšími zariadeniami pripojenými k systému. Pri DMA prenose dát, tieto dáta neprechádzajú priamo cez procesor, ktorý môže v tom čase spracovávať ďalšie inštrukcie. Intel Xeon Phi podporuje DMA. V rámci Intel Xeon Phi koprocesora sú k dispozícií nasledujúce DMA prenosy:

- Intel Xeon Phi koprocesor → Intel Xeon Phi koprocesor GDDR5 pamäť

- Intel Xeon Phi koprocesor GDDR5 pamäť → operačná pamäť hostiteľského systému
- Operačná pamäť hostiteľského systému → Intel Xeon Phi koprocesor GDDR5 pamäť
- Intra-GDDR5 blokový prenos v rámci Intel Xeon Phi koprocesora

Intel Xeon Phi disponuje celkovo ôsmimi DMA kanálmi (angl. DMA channels) pracujúcimi zároveň s nezávislými hardvérovými ring bufframi (ktoré sa môžu nachádzať v pamäti koprocesora alebo v pamäti hostiteľského systému). DMA prenos môže iniciovať ľubovoľná strana a taktiež koprocesor podporuje generovanie prerušenia po dokončení DMA prenosu.

2.2 Programovacie modely

Typická platforma pozostáva z Intel Xeon procesora v kombinácií s Intel Xeon Phi koprocesorom. Prepojením viacerých takýchto konfigurácií môže tvoriť výpočtový klaster. V rámci jedného výpočtového uzla máme niekoľko možností ako spustiť kód na akcelerátore. Tieto možností sú nasledovné:

- Offload model
- Symetrický model
- Natívny model

Tieto módy sú popísané v nasledujúcej časti. Na rozdistribuovanie úloh v rámci klastra sa používa MPI (Message passing interface). Intel Xeon Phi priamo podporuje MPI, možnosti využitia sú popísané taktiež v nasledujúcej časti.

2.2.1 Natívny model

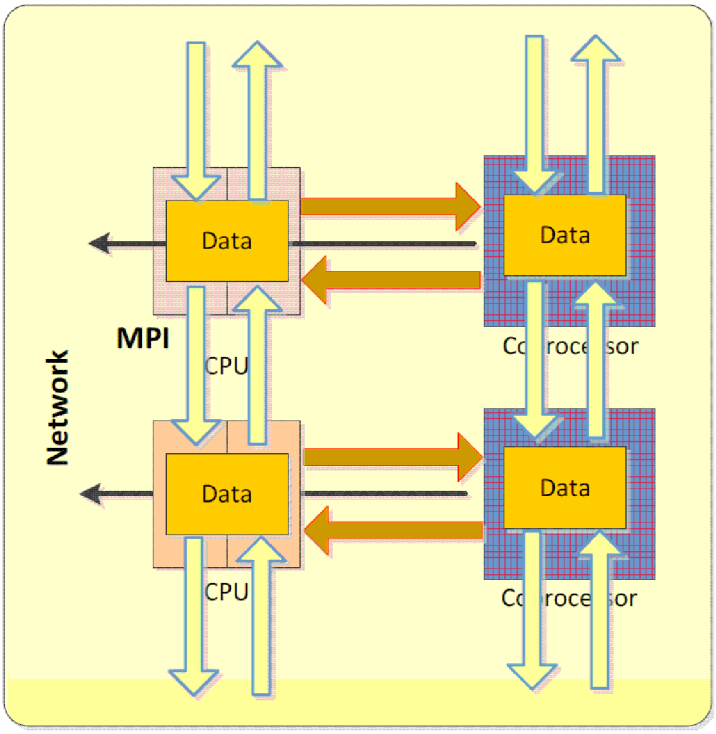
Natívny model umožňuje spúšťať programy priamo na koprocesore. Tieto programy je možné naprogramovať v jazyku C,C++ alebo Fortran a je potrebný kompilátor od Intelu. Pre vytvorenie spustiteľného programu na koprocesore, stačí použiť príslušný prepínač kompilátora. Pre C/C++ ide o prepínač *-mmic*.

Pri vytváraní natívnych programov pre je možné použiť existujúce, dobre známe technológie. Napríklad OpenMP alebo Intel Cilk Plus. Intel taktiež ponúka knižnicu Intel Math Kernel Library, ktorá je optimalizovaná aj pre koprocesor.

Je niekoľko spôsobov ako spustiť natívne program na koprocesore. Keďže na koprocesor je možné sa prihlásiť pomocou SSH protokolu, je s koprocesorom možné pracovať ako s bežným linuxovým počítačom. Spúšťanie programov týmto spôsobom vyžaduje správne nastavenie ciest k dynamickým knižniciam prostredníctvom systémovej premennej *LD_LIBRARY_PATH*. Ďalšou možnosťou je využitie programu *micnativeloadex*, ktorý je súčasťou softvérovej distribúcie Intel Xeon Phi koprocesora. Tento nástroj nakopíruje program, ktorý mu je predaný cez parameter.

2.2.2 Offload model

Pri použití offload modelu, je aplikácia spustená na hostiteľskom procesore. Koprocesor sa používa na akceleráciu niektorých častí programu. Najčastejšie ide o časti, ktoré je možné dobre paralelizovať. Z pohľadu programátora, stačí označiť úsek kódu, ktorý chceme offloadovať na koprocesor, pomocou špeciálnych direktív kompilátora. OpenMP v najnovších



Obr. 2.3: MPI symetrická komunikácia pri behu MPI aplikácií na procesore a koprocesore.

verziách tiež podporuje offload na rôzne koprocesory a akcelerátory. Intel Math Kernel knižnica dokáže automaticky offloadovať niektoré náročné výpočty.

MPI aplikácie bežiace v tomto móde sú spustené na procesore hostiteľského systému, ktorý následne offloaduje niektoré výpočty na koprocesor.

2.2.3 Symetrický model

Symetrický model sa využíva v kontexte MPI aplikácií. Procesor hostiteľského počítača a koprocesor sú z pohľadu MPI rovnocenné uzly. MPI aplikácie bežia jednak na procesor ale tiež na koprocesore. V tejto konfigurácií je potrebné aplikáciu skompilovať dvakrát - pre MIC a pre CPU. Aplikácia beží na koprocesore natívne.

Tento model je najflexibilnejší zo všetkých. Komunikácia môže prebiehať v rámci koprocesora, v rámci procesora, medzi koprocesorom a hostiteľským procesorom, alebo medzi zariadeniami na ostatných uzloch. Obrázok 2.2 ukazuje schému symetrickej komunikácie.

Kapitola 3

Superpočítač Salomon

Počas písania tejto práce sme pracovali so superpočítačom Salomon, ktorý spravuje Národné superpočítačové centrum IT4Innovations v Ostrave. V čase písania tejto práce ide o 48. najvýkonnejší superpočítač na svete (podľa top500.org - November 2015).

"Salomon klaster pozostáva z 1008 výpočtových uzlov, z čoho 432 uzlov obsahuje Intel Xeon Phi koprocesory. Každý uzol je výkonný x86-64 počítač, vybaveným dvomi 12 jadrovými procesormi a 128 GB pamäte RAM. Uzly sú prepojené vysoko rýchlostnými sieťami InfiniBand a Ethernet. Topológia siete je 7D Enhanced hypercube. Všetky uzly zdieľajú 0,5 PB NFS diskové úložisko dostupné pre súbory používateľov. Okrem tohto úložiska, je k dispozícií 1,69 PB DDN Lustre zdieľane úložisko. Užívatelia sa pripájajú ku klastru pomocou štyroch login uzlov."[8] Celý klaster beží pod operačným systémom CentOS. Teoretický výkon celého klastra je 2011 Tflop/s.

Hardvérová špecifikácia výpočtových uzlov s koprocesorom

- Procesor: 2x Intel Xeon E5-2680v3, 2,5GHz (24 jadier)
- Maximálny (peak) výkon procesoru: 19,2 na jadro
- Ram: 128GB, DDR4@2133 MHz
- Pamäťová priepustnosť na úrovni procesora: 68 GB/s
- Topológie siete: InfiniBand FDR56 / 7D Enhanced hypercube
- Koprocesor: 2x Intel Xeon Phi 7120P

Hardvérová špecifikácia Intel Xeon Phi 7120P

- Počet jadier: 61
- Taktovacia frekvencia: 1,238 GHz (1,333 Turbo Boost)
- Maximálny (peak) výkon: 18,4 na jadro
- Cache: 30,5 MB L2
- Pamäťová priepustnosť na úrovni procesora: 352 GB/s

3.1 Softvérová výbava

Používatelia klastra majú k dispozícií radu predinštalovaných knižníc a programov. Zahrnuté sú základné vývojové nástroje, nástroje určené na ladenie programov a tiež nástroje pre vedecké výpočty. Kompletný zoznam softvérovej výbavy je možné nájsť v IT4I dokumentáciu klastra Salomon[1].

Na správu programov klaster využíva EasyBuild. Z pohľadu používateľa, EasyBuild prináša nutnosť načítať moduly príslušných predinštalovaných softvérových balíčkov pred prvým použitím po každom prihlásení pomocou príkazu *module load*. Tento príkaz, v prípade potreby, načíta ďalšie balíky, ak daný balík obsahuje nejaké ďalšie závislosti a nastaví príslušné premenné prostredia. Cesty k spustiteľným programom sú pridané do systémovej premennej *PATH*, cesty k dynamickým knižniciam do premennej *LD_LIBRARY_PATH*.

Z kompilátorov je tu možné nájsť GCC (GNU Compiler Collection) vo verzii $4,9$ až $5,3$, Intel C/C++ a fortran kompilátor vo verzii 2013 až 2016. Ďalej je k dispozícií aj Berkley UPC a Clang.

Pre ladenie a optimalizáciu programov je pripravených niekoľko profesionálnych nástrojov od firmy Intel a Allinea. Allinea Forge je kompletný balík programov pre softvérových vývojárov, ktorý umožňuje debugovať, profilovať a tiež ponúka pomoc s optimalizáciou a kompiláciou C,C++ a Fortran aplikácií pre HPC. Priamo podporované sú knižnice OpenMP, MPI a CUDA. Allinea Performance Reports je nástroj umožňujúci získať základné informácie o behu HPC programov. Informácie sú prezentované formou HTML súboru, ktorý obsahuje informácie o vektorizácí, pamäti, MPI a ďalších.

Intel VTune Amplifier ponúka možnosti ladenia výkonu aplikácií. Okrem podpory Intel CPU, tento nástroj podporuje aj GPU a Intel Xeon Phi koprocesor. Intel VTune Amplifier ponúka niekoľko typov analýz behu programov. Podporované sú OpenMP a tiež MPI. Programátor môže použiť Intel Advisor pri analýze potenciálnych možnosti vektorizácie a paralelizácie na úrovni vlákien. Pomocou tohto nástroja je tiež možné overiť, či príslušné cykly boli vektorizované. Pre analýzu MPI komunikácie medzi programami je k dispozícií Intel Trace Analyzer and Collector.

[1] https://docs.it4i.cz/salomon/software

Kapitola 4

Optimalizačné techniky

Podobnosť architektúry Intel Xeon Phi koprocesorov a Intel Xeon procesorov značne uľahčuje portovanie existujúcich a vytváranie nových softvérových riešení. Je možné využiť skúsenosti s programovaním a optimalizáciou programov pre Intel Xeon procesory. Pri vytváraní aplikácií pre Intel Xeon Phi je odporúčané najprv implementovať a plne optimalizovať túto aplikáciu pre Intel Xeon procesor[9].

Koprocesor disponuje niekoľko násobne vyšším počtom jadier, avšak pri nižšej taktovacej frekvencii. Preto pre dosiahnutie optimálneho výkonu je nevyhnutné efektívne využitie vektorových jednotiek a vysokého počtu hardvérových vlákien.

V nasledujúcich podkapitolách budú predstavené niektoré optimalizačné techniky, ktoré je možné použiť pri vytváraní a optimalizácií aplikácií pre Intel Xeon Phi koprocesor, ale taktiež aj pre Intel Xeon procesor. Táto kapitola obsahuje aj optimalizácie, ktoré sú špecifické pre koprocesor. Niektoré optimalizácie v tejto kapitole sú kompatibilné len s Intel C/C++ kompilátorom. Ďalej táto kapitola poskytuje techniky pre optimalizáciu MPI aplikácií na úrovni zdrojového kódu a tiež na úrovni MPI behového prostredia (angl. runtime).

4.1 Vektorizácia

Vektorizácia je proces, pri ktorom sú skalárne operácie transformované na vektorové operácie. Skalárnymi operáciami sú myslené operácie (inštrukcie) ktoré dokážu spracovať naraz len jeden, respektíve 2 operandy. Vektorové operácie umožňujú spracovať naraz niekoľko párov operandov. Túto transformácia dokáže, niekedy s menšou pomocou, vykonať priamo kompilátor.

Kompilátor vyžaduje zapnutie optimalizácií (prepínač *-On*) a taktiež zapnutie použitia príslušnej SIMD inštrukčnej sady (SSE, AVX alebo IMCI). Použitie prepínača *-xhost* zapne podporu všetkých podporovaných SIMD inštrukčných sád. V prípade kompilácie pre Intel Xeon Phi na Intel Xeon procesore je potrebné použiť prepínač *-mmic*.

Vektorizácia sa často využíva na zrýchlenie spracovania (for) cyklov v programe. Použitím vektorových inštrukcií je možné spracovať naraz niekoľko iterácií (pri jednoduchých cykloch). Dátové závislosti medzi iteráciami cyklu môžu spôsobiť problémy s vektorizáciou. V niektorých prípadoch je možné upraviť spracovanie dát v cykle, čím sa čiastočne dá predísť dátovým závislostiam. V niektorých prípadoch kompilátor nedokáže správne identifikovať dátové závislosti a to spôsobí zlyhanie vektorizácie. Ako tomu predísť je popísane ďalej.

4.1.1 Manuálna vektorizácia

Manuálne je vektorizáciu možné vynútiť niekoľkými spôsobmi. Intel C/C++ poskytuje neštandardný príkaz *#pragma SIMD*, ktorý vynúti vektorizáciu cyklu ktorý nasleduje za týmto príkazom. Vektorizácia pomocou *#pragma SIMD* ignoruje niektoré požiadavky, ktoré musia byť splnené, aby kompilátor mohol automaticky vektorizovať daný cyklus. Kompilátor nekontroluje aliasing a ani dátové závislosti. Manuálna vektorizácia môže zlyhať v prípade, že počet iterácií cyklu nie je známy na začiatku vykonávania cyklu alebo ak cyklus obsahuje podmienene skoky ako *break*. Ak je spolu s týmto príkazom použitá klauzula *ASSERT*, kompilácia zlyhá ak nedôjde k úspešnej vektorizácií. Toto správanie viedlo k prezývke "vectorize or die"pragma.[5]

Alternatívou k skôr spomenutému *#pragma SIMD* je príkaz *#pragma omp simd* zo štandardu OpenMP[11]. OpenMP okrem možností paralelizácie častí programu pomocou vlákien, ktoré sú popísané v časti 4.6, ponúka aj možnosť vektorizácie kódu. Použitím príkazu *#pragma omp simd* pred for cyklom zabezpečí vektorizáciu daného cyklu. Výhodou oproti *#pragma SIMD* je podpora v ostatných kompilátoroch podporujúcich OpenMP štandard. Ďalej ponúka možnosť jednoduchej a efektívnej implementácie vektorizovanej redukcie použitím doplnkovej klauzuly *reduction(operator:list)*.

4.2 Aliasing pamäte

Aliasing pamäte predstavuje stav, kedy do rovnakej časti pamäte ukazuje viacero ukazovateľov (angl. pointer). Implicitne kompilátor predpokladá, že k aliasingu pamäte môže dôjsť, čo môže spôsobiť problémy pri niektorých optimalizáciách. Aliasing je jednou z príčin, prečo môže automatická vektorizácia zlyhať.

Je niekoľko spôsobov ako naznačiť kompilátoru, že pri určitých premenných nedochádza k aliasingu. V jazyku C na to slúži kľúčové slovo *restrict*, ktoré sa používa pri deklarácií parametrov funkcie. Toto kľúčové slovo napovie kompilátoru, že k danej pamäti je možné pristúpiť len pomocou tohto ukazovateľa alebo od ukazovateľov priamo od neho odvodených. Ďalšími spôsobmi sú použitie prepínača kompilátora alebo pragma direktívy *#pragma ivdep*, ktorá je špecifická pre Intel kompilátor.

Táto optimalizácia kompilátoru poskytne informácie, na základe ktorých môže lepšie vektorizovať cykly a optimalizovať vygenerovaný kód.

4.3 Zarovnanie pamäte

Vektorové inštrukcie sú vykonávané efektívnejšie, ak pamäť s ktorou pracujú sú zarovnané na príslušnú veľkosť. Zarovnanie pamäte znamená, že adresa pamäte je násobkom konkrétnej hodnoty. V prípade AVX2 inštrukčnej sady (Intel Xeon procesor), je táto hodnota 32 bajtov. Inštrukčná sada IMCI koprocesora vyžaduje zarovnanie pamäte na 64 bajtov. Taktiež je výhodné použiť zarovnanie podľa veľkosti cache line. Alokácia zarovnanej pamäte je možná pomocou napríklad funkcie *_mm_malloc* (neštandardná funkcia, Intel kompilátor) alebo pomocou C++11 štandardnej funkcie *aligned_alloc()*. Statické premenné alebo polia je zas možné deklarovať v C++11 pomocou špecifikátora *alignas(n)*. Pri dynamicky alokovanej pamäti si kompilátor nevedie záznam o zarovnaní a preto je niekedy potrebné manuálne informovať kompilátor o zarovnaní pred použitím. Intel kompilátor na to využíva konštrukciu *__assume_aligned(x,size)*. K dispozícií sú ďalšie alternatívy. Ďalšou špecifickou pre Intel kompilátor je pragma direktíva *#pragma vector aligned*, ktorá sa používa

pred cyklom. Táto direktíva spôsobí, že kompilátor bude predpokladať, že všetky polia sú zarovnané. Pri použití manuálnej vektorizácie pomocou OpenMP, je možné použiť klauzulu *aligned*.

Zarovnanie pamäte sa spája so začiatkom pamäťového bloku. Taktiež je dôležité, aby veľkosť bloku pamäte, respektíve riadku v poli bola zarovnaná na určitý počet bajtov. V tomto kontexte ide o takzvaný padding. Vo väčšine prípadov ide o rovnakú hodnotu ako pri zarovnaní pamäte. Toto je dôležité použiť napríklad pri maticiach, ktoré sú v pamäti uložené lineárne v jednom bloku pamäte. Bez použitia paddingu sú začiatky riadkov, respektíve stĺpcov (závislé na interpretácii - row major/column major) nezarovnané.

Ak program nepracuje so zarovnanou pamäťou, pri vektorizácií dochádza k vygenerovaniu takzvaných peel a remainder cyklov. Sú to cykly, ktoré pozostávajú zo skalárnych inštrukcií a spracujú časť cyklu tak, aby vektorizovaná časť cyklu pracovala so zarovnanou pamäťou. Peel cyklu sa dá zbaviť použitím zarovnanej pamäte.

4.4 Usporiadanie dátových štruktúr

Pri optimalizácií algoritmov, reprezentácia dátových štruktúr v pamäti hrá dôležitú úlohu. Klasický prístup spočíva v umiestnení dát do štruktúr. Kód 4.1 zobrazuje príklad takejto štruktúry, ktorá reprezentuje bod v priestore. Často algoritmus vyžaduje väčšie množstvo takýchto štruktúr. Prirodzené je vytvoriť pole štruktúr (angl. array of strucutres - AoS). To však môže spôsobovať problémy pri vektorizovaní spracovania takýchto dát. V pamäti sa jednotlivé atribúty štruktúr nachádzajú lineárne za sebou. V niektorých prípadoch kompilátor môže doplniť padding, zväčša pri kompozícií rozdielnych typov s rozdielnou veľkosťou. Vektorizácia cyklov, ktoré spracovávajú dáta v takejto forme môže byť problémová.

```
struct point
{
    float x;
    float y;
    float z;
};
point points[42];
```

Listing 4.1: Pole štruktúr

Pri použití vektorizácie je takmer stále lepšie použiť reprezentáciu dát vo forme štruktúry polí (angl. structure of arrays - SoA)[1]. Táto forma spočíva v uložení jednotlivých atribútov štruktúry do poľa priamo v štruktúre, ako to zobrazuje úryvok 4.2. Spolu s touto formou reprezentácie dát je vhodné tiež použiť zarovnanie pamäte. V niektorých prípadoch, dokáže kompilátor jednoduchšie a efektívnejšie vektorizovať cykly využívajúce takúto formu dát z pamäti. Napríklad pri algoritme, ktorý vykonáva transformáciu jednotlivých bodov nezávisle na ostatných, s použitím vektorizácie je možné spracovať niekoľko bodov naraz (závislé na veľkosti SIMD registrov a použitých dátových typoch). Čítanie a zápis sú v takom prípade efektívne, keďže je použitý zarovnaný prístup do pamäte. V každej iterácií je niekoľko atribútov z každého poľa načítaných do SIMD registrov, nad ktorými sú následne vykonané vektorové operácie. Pri použití AoS to nie je príliš možné.

```
struct soa_points;
{
    float x[42];
    float y[42];
    float z[42];
};
soa_points points;
```

Listing 4.2: Pole štruktúr

4.5 Streaming stores

Streaming stores je technika, špeciálne navrhnutá pre zvýšenie pamäťovej šírky pásma (angl. memory bandwith) a na zvýšenie využitia cache pamäte. Streaming stores sú zápisy do pamäte, ktoré nevyžadujú čítanie danej pamäte pri zápise. Typicky je čítanie pred zápisom štandardné správanie, ale v prípadoch, kde nie je čítanie pred zápisom potrebné, čítanie zaberá dodatočnú pamäťovú šírku pásma. Intel Xeon Phi podporuje streaming stores na úrovní inštrukcií, teda je nevyhnutné aby kompilátor vygeneroval príslušné inštrukcie.[9]

Príklad 4.3 zobrazuje cyklus for, v ktorom je výhodné použiť streaming stores. Bez použitia streaming stores sú dáta z poľa A najprv prečítané do cache pamäte a následne sú prepísané. Celkovo sú vykonané štyri pamäťové operácie - prečítanie polí A,B,C a zápis A. Pri použití streaming stores je počet operácií zredukovaný na tri - prečítanie B,C a zápis A, čím bola ušetrená pamäťová šírka pásma.

```
for (i=0; i<HUGE; i++)
    A[i] = K*B[i] + C[i];
```

Listing 4.3: Prípad, v ktorom je výhodné použiť streaming stores

Streaming stores sa používajú zväčša spolu s vektorizáciou. Kompilátor dokáže použiť streaming stores v niektorých prípadoch automaticky, čo vyžaduje zarovnaný vektorový zápis do pamäte bez použitia maskovania a musí byť zapísaná celá cache line ($64B$). Kompilátor môže tiež vygenerovať kód, ktorý dynamicky spracuje niekoľko začiatočných iterácií aby ďalej mohol vykonávať pamäťové operácie nad zarovnanými adresami (tzv. peel loop). Pre manuálne vynútenie použitia streaming stores na zápis dát v cykle je možné použiť pragma direktívu *#pragma vector nontemporal*.

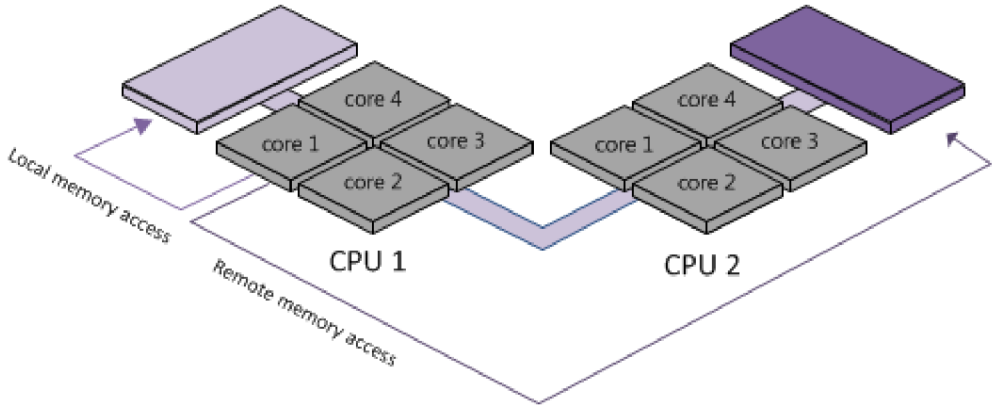
4.6 Paralelizácia pomocou OpenMP

Predchádzajúce optimalizácie sa zameriavali predovšetkým na zvýšenie výkonu v rámci jedného vlákna. Pre dosiahnutie maximálneho výkonu je ďalej potrebné využiť všetky dostupné jadrá procesora, respektíve koprocesora. Pomocou OpenMP je možné použitím niekoľkých pragma direktív na správnych miesta paralelizovať program pomocou viacerých vlákien.

Príklad 4.4 zobrazuje cyklus for paralelizovaný pomocou OpenMP. Pred spracovaním tohto cyklu, je vytvorená skupina vlákien (*#pragma omp parallel*), medzi ktorú sú následne rozdelené iterácie cyklu (*#pragma omp for*). V tomto prípade je práca rozdelená pravidelne

```
#pragma omp parallel for shared(A,B,C,k) private(i)
for (i=0; i<N; i++)
    A[i] = K*B[i] + C[i];
```

Listing 4.4: Použitie OpenMP na paralelizovanie for cyklu



Obr. 4.1: NUMA systém s dvomi NUMA uzlami.

medzi všetky dostupné vlákna a každé vlákno má pridelený rovnaký počet iterácií *schedule(static)*.

Počet OpenMP vlákien je možné nastaviť niekoľkými spôsobmi. Jedným z nich je nastavením premennej prostredia *OMP_NUM_THREADS* s ktorou je možné jednoducho meniť počet vlákien bez nutnosti zmeniť zdrojový kód. OpenMP ďalej umožňuje špecifikovať ako budú vlákna rozložené na jednotlivých jadrách a ich dostupných hardvérových vláknach. Možnosti sú *compact* a *scatter*. Prvé umiestni vlákna na hardvérové vlákna zaradom (prvé štyri vlákna sú umiestnené na 1. jadro koprocesora, ďalšie štyri vlákna na 2. jadro koprocesora atď.). Scatter rozmiestni vlákna rovnomerne po všetkých dostupných jadrách. Taktiež je možné vlákna naviazať na jadro, hardvérové vlákno alebo socket. Tieto parametre je taktiež možné meniť pomocou premennej prostredia.

4.7 NUMA First Touch Policy

Výpočtové klastre často pozostávajú z viac-procesorových výpočtových uzlov. Tieto viac-procesorové systémy využívajú NUMA (Non-Uniform Memory Access). V tomto systému, procesory sú zoskupené spolu s operačnou pamäťou do tzv. NUMA uzlov. Každý takýto uzol má vlastný procesor, operačnú pamäť a zbernicu. NUMA uzly sú tiež prepojené zbernicou. Táto zbernica je ale pomalšia ako zbernica medzi procesorom a pamäťou v rámci uzlov. Ak sa dáta, ku ktorým chcem procesor pristupovať, nachádzajú v inom NUMA uzle, ako v tom, v ktorom sa nachádza tento procesor, tieto dáta musia byť prenesené po zbernici, ktorá spája tieto NUMA uzly. Teda prístup k týmto dátam je pomalší, ako keby sa nachádzali v lokálnej pamäti. Z toho názov Neuniformný prístup k pamäti. Na obrázku[1] 4.1 je znázornená schéma dvoch NUMA uzlov prepojených zbernicou.

Umiestnenie pamäťových stránok (angl. memory pages) je v súčasných operačných systémoch riadené tzv. First Touch Policy. Po alokácií bloku pamäte programom, sa stránka

[1] http://frankdenneman.nl/2010/09/13/esx-4-1-numa-scheduling/

obsahujúca túto pamäť reálne umiestni do fyzickej pamäte až pri inicializácií (prvom zápise) do tejto pamäte. Stránka je umiestnená do pamäte v rámci NUMA uzla, ktorého procesor prvý zapísal dáta do tejto stránky.

Toto správanie môže viesť k nežiadúcemu poklesu výkonu pri paralelizácií niektorých algoritmov, konkrétne v prípade ak sú všetky potrebné dáta inicializované jedným vláknom a pri ďalšom spracovaní k týmto dátam pristupujú aj vlákna, ktoré sú umiestnené na inom NUMA uzle ako vlákno, ktoré inicializovalo danú pamäť.

Riešením tohto problému je tzv. NUMA First Touch Policy. Spočíva v paralelnej inicializácií dát, čo spôsobí rozloženie dát medzi NUMA uzlami. Táto technika sa často používa v spojení s paralelizáciou pomocou OpenMP.

4.8 Message Passing Interface

Zasielanie správ (angl. Message passing) je technika používaná pri programovaní paralelných aplikácií distribuovaných na viacero počítačov, respektíve aplikácií pre klastre, ktoré bežia na mnoho uzloch. Komunikácia medzi výpočtovými jednotkami prebieha pomocou zasielania správ. Túto techniku je možne použi v dvoch typoch počítačových architektúr podľa Flynovej taxonómie - SPMD (jeden program, viacero dát) a MPMD (viacero programov, viacero dát). Jeden program, viacero dát (angl. Single Program Multiple Data) predstavuje systém, v ktorom beží nezávisle na sebe niekoľko inštancií rovnakého programu, pričom každý má vlastný kontext. Každá inštancia väčšinou pracuje nad inou množinou dát ako ostatné. Viacero programov, viacero dát (angl. Multiple Programs Multiple Data) je systém podobný SPMD. Rozdielom je, že pri MPMD jednotlivé inštancie programov nie sú spustené z toho istého programu.

Štandardizovaným systémom využívajúcim túto techniku je Message Passing Interface[2] (ďalej len MPI). MPI špecifikuje rozhranie funkcií potrebných pre zabezpečenie komunikácie medzi procesmi. Jadrom MPI sú komunikačné funkcie. Podporované sú blokujúce aj neblokujúce (a ďalšie variácie) komunikácie. Existuje niekoľko implementácií, ktoré tento štandard implementujú. Táto práca je postavená nad implementáciou od firmy Intel.

4.8.1 Prekrývanie komunikácie s výpočtom

Blokujúca komunikácia spôsobuje plytvanie času. Pri čakaní na dokončenie komunikácie vlákno nevykonáva žiadnu užitočnú prácu. V niektorých prípadoch, ak to dátové závislosti dovoľujú, je možné prekryť komunikáciu a výpočty. Prenos dát je inicionvaný pred začatím výpočtu, ktorý tieto dáta nevyužíva. Paralelne s výpočtom prebieha komunikácia medzi procesmi. Použitím neblokujúcich komunikácií vzniká potreba manuálnej synchronizácie. Programátor musí zabezpečiť, aby program nepristupoval do pamäte, ktorú využíva aktívna komunikácia.

4.8.2 Nastavenia behového prostredia

Implementácia MPI od firmy Intel poskytuje možnosť zmeniť niektoré parametre MPI behového prostredia. Tieto nastavenia je možné zmeniť pomocou systémových premenných.

V štandardnej konfigurácií je vypnutá podpora pre beh MPI na Intel Xeon Phi koprocesore. Nastavením systémovej premennej *I_MPI_MIC* na hodnotu 1 je možné spúšťať MPI procesy priamo na koprocesore použitím príslušných parametrov v príkaze *mpirun*.

[2] http://www.mpi-forum.org/docs/docs.html

Pre dosiahnutie optimálneho výkonu v rámci Salomon klastra, je odporúčané[7] použiť konfiguráciu, ktorá je v úryvku 4.5. Tieto nastavenia zabezpečia, že komunikácia v rámci výpočtového uzlu bude používať *SHMEM* komunikáciu medzi procesorom a koprocesorom, na komunikáciu sa použije *IB SCIF* rozhranie a medzi výpočtovými uzlami bude komunikácia prebiehať cez *CCL-Direct proxy*.

```
export I_MPI_FABRICS=shm:dapl
export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u,ofa-v2-scif0 , \
                                ofa-v2-mcm-1
```

Listing 4.5: Nastavenia behového prostredia Intel MPI pre optimálny výkon v rámci klastra Salomon.

Ďalším odporúčaným[6] nastavením behového prostredia pri použití OpenMP spolu s Intel MPI je vytvorenie naviazania medzi MPI procesmi a OpenMP vláknami. Toto nastavenie je použité ak je nastavená systémová premenná *I_MPI_PIN_DOMAIN* na hodnotu *omp*.

Kapitola 5

N-Body problém

Pre demonštráciu optimalizácií na komplexnejšej úlohe bol zvolený problém N-body simulácie. Klasický N-body problém simuluje vývoj dynamického systému o N telesách (z toho N-body). Každé teleso v systéme ovplyvňuje silou všetky ostatné telesá. Takéto simuláciu majú široké využitie v rôznych oblastiach ako astrofyzika (skúmanie a simulácia pohybu vesmírnych telies) alebo molekulárna dynamika (simulácie molekúl kvapalín a podobne).[2]

V každom kroku simulácie je vypočítaná celková sila pôsobiaca na každú časticu, respektíve teleso v systému a následne je aktualizovaná jej poloha a ostatné atribúty. Najzákladnejšia verzia tejto simulácie počíta interakcie medzi časticami po pároch. Výsledná sila je vypočítaná ako súčet síl medzi danou časticou a všetkými ostatnými po pároch. Zložitosť tohto prístupu je $O(N^2)$. V každom kroku simulácie je potrebných N^2 výpočtov sily.[2]

Pre výpočet pôsobenia gravitačnej sily sa používa Newtonov gravitačný zákon, ktorý zobrazuje rovnica 5.1, kde $\mathbf{F}_1$ je gravitačná sila pôsobiaca na teleso i telesom j, κ je gravitačná konštanta, m_1 a m_2 sú hmotnosti telies, r je vzdialenosť medzi telesami a $\mathbf{r}$ je polohový vektor. V kontexte N-body simulácie je použitý vzťah 5.2, ktorý vyjadruje celkovú silu pôsobiacu na časticu v rámci N-body systému. V tomto vzťahu $\mathbf{r}_{ij}$ predstavuje vektor medzi telesom i a j. Zavedením tzv. zjemňujúceho faktoru (angl. softening factor) je možné odstrániť podmienku $j \neq i$. Zjemňujúci faktor je vhodné použiť ak v systéme nie sú modelované kolízie telies. Rovnica 5.3 zobrazuje vzťah na výpočet gravitačnej sily na teleso s použitím zjemňujúceho faktoru.[10]

$$\mathbf{F}_i = -\kappa \frac{m_i m_j}{r^2} \frac{\mathbf{r}}{r} \tag{5.1}$$

$$\mathbf{F}_i = \sum_{1 \leq j \leq N} \mathbf{f}_{ij} = \kappa m_i \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}, j \neq i \tag{5.2}$$

$$\mathbf{F}_i \approx \kappa m_i \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{(\|\mathbf{r}_{ij}\|^2 + \varepsilon^2)^{\frac{3}{2}}} \tag{5.3}$$

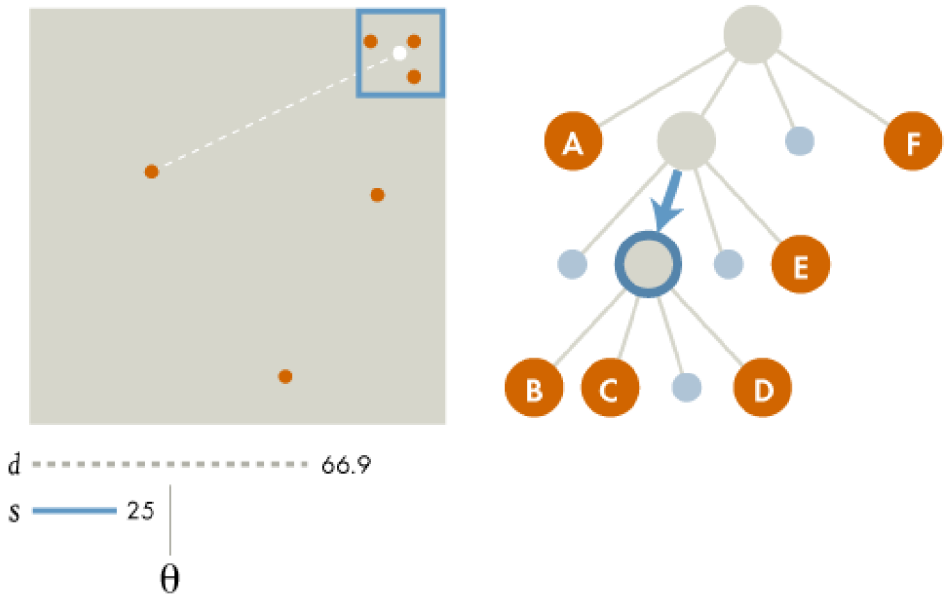
Výpočet novej pozície častice na základe výslednej sily je založený na vzťahu $\mathbf{v} = \mathbf{a}t$ a $\mathbf{F} = m\mathbf{a}$. Simulácia prebieha s konštantným časovým krokom. Predchádzajúce vzťahy a vzťah 5.3 je možné ďalej upraviť tak, aby vo výpočtoch nebolo potrebné uvažovať akceleráciu a hmotnosť častice i.

5.1 Barnes-Hut

Značnú časť výpočtu N-body simulácie v predchádzajúcej časti zaberá výpočet síl na častice. Preto boli vyvinuté metódy, ktoré sa snažia urýchliť výpočet interakcií medzi časticami. Jednou z takýchto metód je Barnes-Hut algoritmus. Tento algoritmus je na rozdiel od naivnej verzie riešenia N-body problému komplexnejší a preto bol zvolený na demonštráciu niektorých optimalizácií.

Základom tohto algoritmu je hierarchická reprezentácia priestoru pomocou oktálového stromu (angl. octree). Pre výpočet síl, je ako prvé, potrebné zostaviť okálový strom z častíc v systéme a v jednotlivých uzloch stromu vypočítať celkovú hmotnosť a ťažisko všetkých častíc, ktoré sa nachádzaj podstrome daného uzlu.

V ďalšej fáze algoritmu je vypočítaná celková sila pôsobiaca na časticu počas priechodu stromom. Pre každú časticu je strom prejdený raz. Strom sa prehľadáva do hĺbky a začiatok priechodu je v koreni stromu. V každom kroku, ak častica, pre ktorú počítame celkovú silu, je dostatočne ďaleko od ťažiska aktuálneho uzlu, je použité ťažisko a súčet hmotností daného uzlu. Táto operácia predstavuje aproximáciu sily danej častice so všetkými časticami v podstrome daného uzlu. Tento podstrom sa ďalej už neprehľadáva. V opačnom prípade, ak je častica príliš blízko ťažiska uzlu, je potrebné pokračovať v priechode stromom pre všetkých osem podstromov daného uzlu. Uzol je považovaný za dostatočne vzdialený od častice, ak pomer jeho veľkosti (dĺžka strany obalovej kocky) a vzdialenosti jeho ťažiska od častice je menší ako parameter θ. Asymptotická zložitosť tohto algoritmu je $O(N \log N)$. Pri nízkych hodnotách parametra θ tento algoritmus môže degradovať na $O(N^2)$.Parameter θ taktiež ovplyvňuje chybu výpočtu celkovej sily, kedy zvyšujúca sa hodnota parametra tiež zvyšuje chybu.[2]. Obrázok[1] zachytáva schému testovacieho kritéria.



Obr. 5.1: Testovanie, či sa pre daný uzol (modré rámovanie) použije pre výpočet sily ťažisko tohto uzlu (biely bod) alebo sa bude pokračovať v prechode stromom do hĺbky v danom uzle. Porovnáva sa šírka príslušného uzlu a vzdialenosť ťažiska od bodu pre ktorý počítame silu.

[1] http://frankdenneman.nl/2010/09/13/esx-4-1-numa-scheduling/

Kapitola 6

Výkonnostné Testy

Pri návrhu, implementácií a optimalizácií aplikácií je vhodné, sa ako prvé, zoznámiť s daným systémom, pre ktorý je určená aplikácia. Táto kapitola má za cieľ analyzovať dostupný výkon výpočtového klastra. Všetky hodnoty výkonu v tejto práci predpokladajú využitie operácií nad 32-bitovými desatinnými číslami (4B float).

6.1 Násobenie Matice a Vektora

Tento jednoduchý výkonnostný test pozostáva z niekoľkonásobného opakovania násobenia matice a vektora. Príklad 6.1 zobrazuje pseudokód násobenia matice a vektora.

```
for(i = 0; i < ROWS; i++)
    for(j = 0; j < COLS; j++)
        o[i] += m[i][j] * v[j];
```

Listing 6.1: Pseudokód násobenia matice a vektora

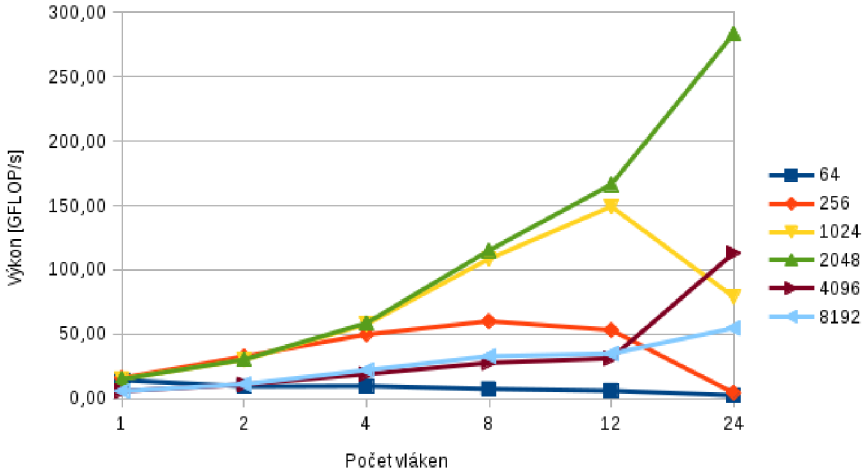
Cieľom tohto benchmarku je snaha o priblíženie sa k maximálnemu výpočtovému výkonu, ktorý procesor, respektíve koprocesor môže dosiahnuť. Beh tohto testu pozostáva z 1000 opakovaní násobenia matice a vektora. Veľkosti a taktiež počet vlákien boli postupne zvyšované. V teste programe boli využité niektoré optimalizačné techniky z kapitoly 4.

Obrázky 6.1 a 6.2 zobrazujú poslednú verziu nameraných výsledkov.

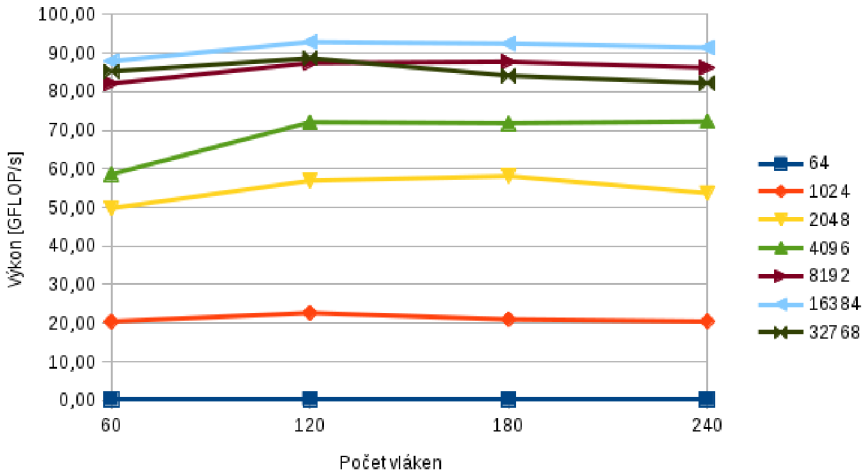
Zhodnotenie Pri 1000 opakovaniach, maximálny výkon na procesore bol dosiahnutý pri vstupnej matici o veľkosti 2048x2048 (16 MiB) a to na úrovni 280 GFLOP/s. V prípade akcelerátora, maximálny výkon predstavoval 90 GFLOP/s pri veľkosti vstupnej matice 16384x16384 (1 GiB).

Výkon na CPU do značnej miery rástol so zvyšujúcim sa počtom vlákien. Pri vyšších vstupoch (> 2048) môžeme pozorovať degradáciu výkonu. Výkon algoritmu na koprocesore s rastúcim počtom vlákien sa príliš nemenil.

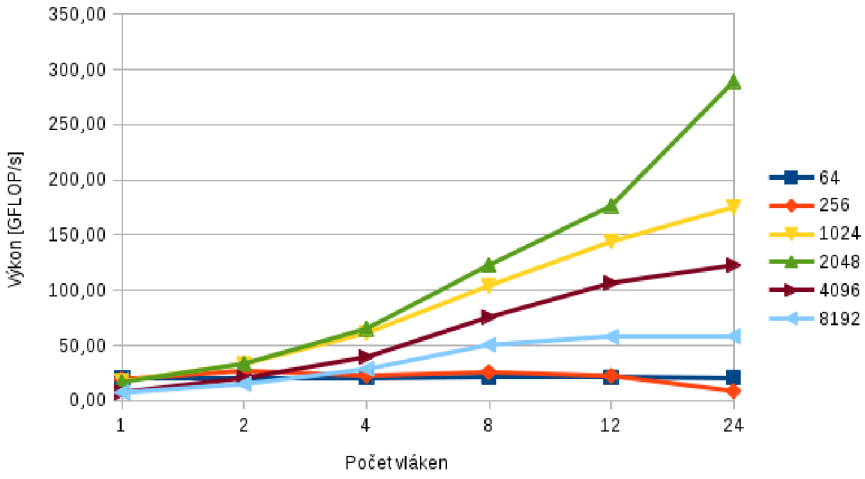
Tento test meral čas opakovaného volania funkcie, ktorá vykonávala násobenie a matice. V tejto funkcii boli použité OpenMP konštrukcie. Nízky výkon dosiahnutý v tomto výkonnostnom teste bol do značnej miery zavinený zahrnutím réžie spojenej s opakovaným vytváraním vlákien.



Obr. 6.1: Matvec na Intel Xeon procesore



Obr. 6.2: Matvec na Intel Xeon Phi koprocesore



Obr. 6.3: Beh Intel MKL funkcie na násobenie matice a vektora na Intel Xeon procesore (1000 opakovaní).

6.2 Násobenie Matice a Vektora - Intel MKL

Knižnica Math Kernel Library[1] (skrátene MKL) od spoločnosti Intel ponúka vysoko optimalizované funkcie lineárnej algebry, rýchlej Fourierovej transformácie, vektorovej matematiky, štatistiky a ďalšie. Tieto funkcie sú manuálne optimalizované pre relevantné Intel architektúry. Podporované sú rozhrania LAPACK a BLAS. Intel MKL tiež podporuje Intel Xeon Phi koprocesor, natívny mód a offload mód.

Tento výkonnostný test slúži na určenie maximálneho možného dosiahnuteľného výkonu pri operácií násobenia matice a vektoru a ako referencie pre porovnanie s násobením matice a vektora implementovaného v tejto práci. Bola použitá funkcia cblas_sgemv.

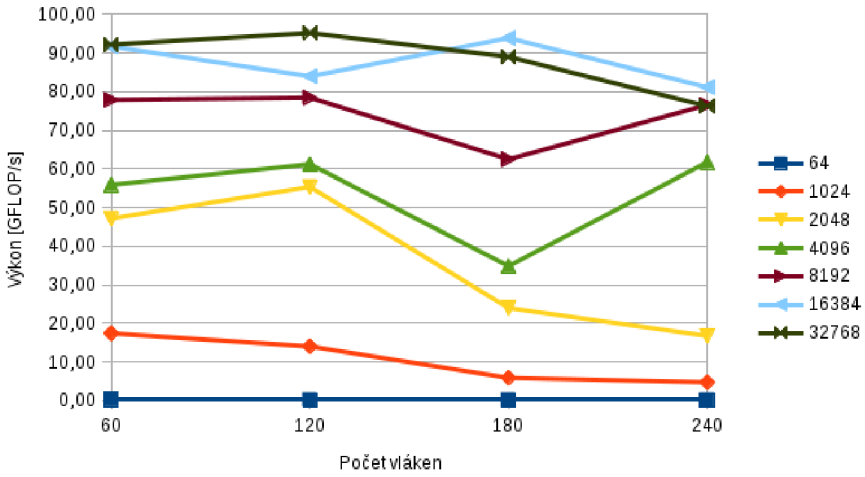
Test prebiehal podobne ako v predchádzajúcom teste. Niekoľkonásobné opakovanie volania funkcie s rôznymi konfiguráciami počtu procesov a veľkostí vstupnej matice a vektora. Merania z tohoto testu sú zachytené na obrázku 6.3 a 6.4.

Zhodnotenie Namerané hodnoty dosahujú podobnú úroveň ako v predchádzajúcom výkonnostnom teste. Maximu verzie na procesore sa pohybuje okolo hodnoty 290 GFLOP/s pri použití 24 vlákien a veľkosti vstupnej matice 2048x2048. Infotel MKL verzia pracuje efektívnejšie pri maximálnom počte vlákien.

Pri behu na Intel Xeon Phi akcelerátore sme zaznamenali výrazné kolísania výkonu pri použití viac ako dvoch vlákien na jedno jadro. Nami implementovaný matvec test disponoval stabilnejším výkonom v tomto prípade.

Rovnako ako v predchádzajúcom prípade, v meraní času behu funkcie bola zahrnutá réžia vytvárania vlákien.

[1] https://software.intel.com/en-us/intel-mkl



Obr. 6.4: Beh Intel MKL funkcie na násobenie matice a vektora na Intel Xeon Phi koprocesore (1000 opakovaní).

Tabuľka 6.1: STREAM benchmark výsledky - CPU

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	119581,0	0,001358	0,001338	0,007264
Scale:	87575,2	0,001853	0,001827	0,013165
Add:	115492,5	0,002115	0,002078	0,013728
Triad:	117941,8	0,002059	0,002035	0,010773

6.3 STREAM Benchmark

STREAM benchmark[2] bol navrhnutý s cieľom určiť tzv. udržateľnú šírku pásma pamäte (angl. sustainable memory bandwith). Benchmark bol skompilovaný s podporou OpenMP a zapnutou optimalizáciou na úrovni 3. Počet vlákien bol nastavený na maximálnu odporúčanú hodnotu, ktorá je podporovaná na danom procesore - 24 vlákien na CPU a 240 na MIC. Veľkosť vstupných polí bola nastavená na 10 miliónov (76,3 MB) a počet opakovaní na 1000. Tabuľky 6.1 a 6.2 obsahujú namerané hodnoty.

[2] https://www.cs.virginia.edu/stream/

Tabuľka 6.2: STREAM benchmark výsledky - MIC

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	156248,8	0,001077	0,001024	0,001896
Scale:	125109,7	0,001344	0,001279	0,001650
Add:	140669,8	0,001785	0,001706	0,002115
Triad:	142179,8	0,001768	0,001688	0,002096

6.4 MPI OSU Benchmark

MPI OSU microbenchmark[3] je syntetický výkonnostný test MPI komunikácie. Nejedná sa o jeden test, ale o balíček nezávislých testov jednotlivých MPI komunikačných funkcií. MPI testy sú rozdelené do dvoch hlavných skupín: testy komunikácie medzi dvomi procesmi a testy kolektívnych operácií. Z každej skupiny sme zvolili niekoľko testov. Kompletný zoznam zoznam všetkých testov je možné nájsť na webovej stránke testu. Z prvej skupiny sme zvolili nasledovné:

- OSU Latency - priemerná doba medzi odoslaním správy procesom a prijatím odpovede
- OSU Bandwidth - maximálna prenosová rýchlosť medzi procesmi na úrovni siete
- OSU Bidirectional Bandwidth - maximálny udržateľný agregovaný bandwidth

Testovaných bolo niekoľko konfigurácií umiestnenia procesov - všetky kombinácie medzi procesormi a koprocesormi v rámci dvoch uzlov. Pri behu výkonnostných testov, bola použitá konfigurácia MPI behového prostredia popísaná v časti 4.8.2. Jeden z grafov obsahujúcich namerané hodnoty OSU výkonnostného testu je na obrázku 6.5. Zvyšné grafy sa nachádzajú v prílohe B.

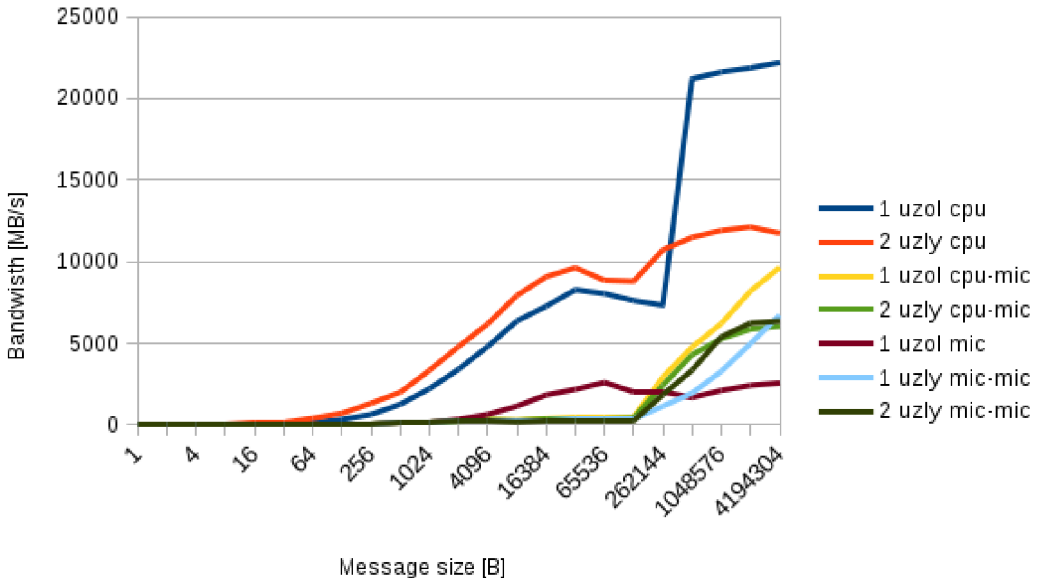
Najvyššia priepustnosť v oboch smeroch bola zaznamenaná pri komunikácií dvoch procesov v rámci jedného procesora. Predpokladáme, že výrazný skok pri veľkosti správy $262144B$ bol zapríčinený vnútornou zmenou stavu MPI implementácie (prepnutie použitej komunikačnej technológie). Obojstranná priepustnosť medzi procesmi bežiacimi na jednom koprocesore bola nižšia ako pri komunikácií procesov bežiacich na procesore a koprocesore. Pri odosielaní správ v jednom smere, došlo k zníženiu priepustnosti vo väčšine prípadov, okrem prípadu komunikácie dvoch procesov v rámci jedného koprocesora. Z úrovne $2500MB/s$ na $3500MB/s$. Priepustnosť komunikácie v rámci jedného procesora dosiahla hodnoty namerané v STREAM výkonnostnom teste, čo značí použitie zdieľanej pamäte.

Výkonnostné testy zamerané na kolektívne operácie merajú latenciu daných komunikačných funkcií. Zvolili sme nasledovné testy:

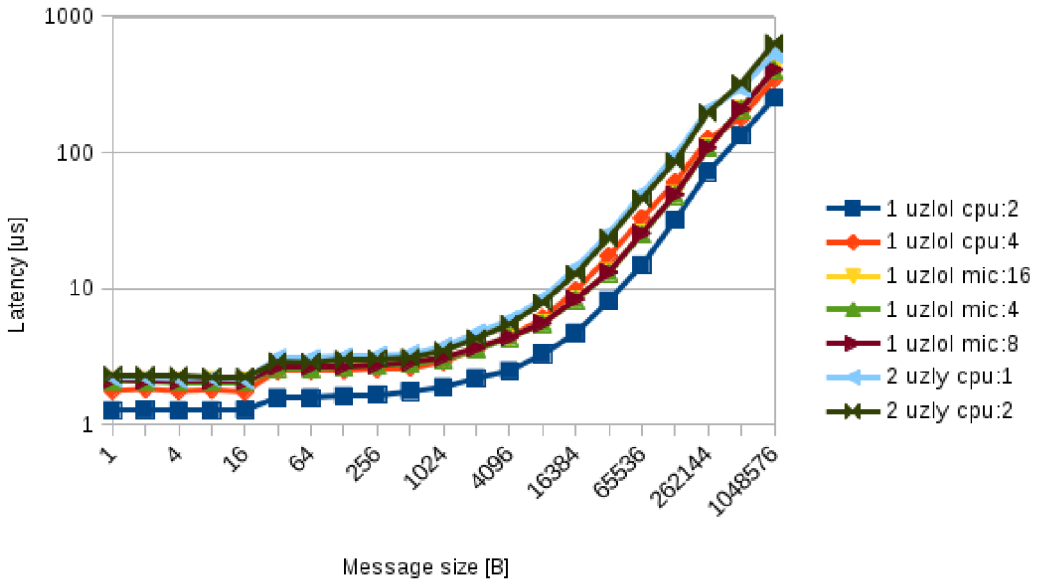
- OSU Scatter
- OSU Bcast
- OSU Reduce
- OSU Allreduce
- OSU Alltoall

Podobne ako pri testoch komunikácie medzi dvomi procesmi bolo použitých niekoľko konfigurácií rozloženia procesov. Maximálny početu zlov na ktorých boli tieto výkonnostné testy spúšťané bol 3.

[3] http://mvapich.cse.ohio-state.edu/benchmarks/



Obr. 6.5: Namerané hodnoty výkonnostného testu OSU Bidirectional Bandwidth medzi dvomi procesmi v rôznych konfiguráciach.



Obr. 6.6: Namerané hodnoty výkonnostného testu OSU Scatter - Latency.

Kapitola 7

Implementácia a Optimalizácia N-body problému

Doposiaľ sme sa zaoberali optimalizáciou len teoreticky. V nasledujúcej kapitole popíšeme proces implementácie a postupnej optimalizácie problému N-Body. Začneme s jednoduchšou verziou tohto problému, s naivnou verziou, v ktorej sa pre výpočet výslednej sily počíta interakcia každého s každým. Následne sa budeme venovať riešeniu N-Body problému s využitím stromu pre akceleráciu výpočtu síl.

Barnes-Hut verzia N-Body problému bola od začiatku navrhnutá ako MPI aplikácia. Najprv bola vytvorená verzia bez použitia OpenMP konštrukcií, ale s využitím MPI funkcií, ktorá slúžila ako základ pre merania dopadu ostatných optimalizácií. Uvažovali sme, že MPI funkcie majú pri behu len jedného procesu minimálny dopad na výkon.

Základné verzie aplikácií vytvorených v rámci tejto práce boli vyvíjané v jazyku C++ pomocou kompilátora GCC 5.3.1. Vo fáze optimalizácie bol použitý kompilátor Intel C/C++ verzie 2016.01. Snahou bolo použiť čo najmenej neštandardných konštrukcií, preto namiesto niektorých Intel špecifických pragma direktív bolo použité OpenMP. Kompilátor GCC avšak nedisponuje schopnosťami optimalizácie kódu na úrovni Intel kompilátora, preto bolo nevyhnutné vo fáze optimalizácie použiť tento kompilátor.

Zdrojové kódy programov implementovaných a optimalizovaných v tejto kapitole je možné nájsť v priloženom CD, alebo tiež v git repozitári na adrese https://bitbucket.org/tomas_k/xeon_phi_benchmarks.

Počas procesu optimalizácie boli použité nástroje dostupné na superpočítači Salomon - Intel VTune Amplifier 2016, Intel Advisor, Intel®Trace Analyzer and Collector a ďalšie nástroje od spoločnosti Allinea.

Metódy verifikácie Pri vytváraní aplikácií nebol kladený dôraz na numerickú korektnosť N-Body simulácie. Na druhú stranu, pri zavedení každej novej optimalizácie, respektíve paralelizmu, bola testovaná korektnosť výstupu oproti základnému prípadu bez paralelizmu a optimalizácií. Porovnávané boli aj výsledné hodnoty medzi CPU a MIC verziami. Výsledné hodnoty boli porovnávané na základe kontrolného súčtu, ktorý bol vypočítaný ako súčet absolútnych hodnôt atribútov všetkých častíc.

Merania Po každej fáze optimalizácie boli vykonané merania doby trvania vykonávania hlavnej časti programu. V priebehu optimalizácie boli zvolené menšie vstupné datasety s menším počtom iterácií, ktoré slúžia ako demonštrácia dopadu jednotlivých optimalizácií na

výkon. V závere, po dosiahnutí optimálnej úrovne optimalizácií, boli použité väčšie datasety na odhalenie škálovania.

Výkon algoritmu v $GFLOP/s$ (10^9 operácií s desatinným číslom za sekundu, angl. Floating Points Operations per Second) bol vypočítaný manuálne, na základe vstupných parametrov a výsledného času. V prípade naivnej verzie táto hodnota bola vypočítaná podľa vzorca 7.1, kde V predstavuje výkonnosť v $GFLOP/s$, N počet častíc, S počet krokov simulácie a t je celkový čas behu simulácie. Konštanty 22 a 15 boli určené ručným spočítaním operácií s desatinnými číslami v príslušných cykloch.

$$V = \frac{S * N * (22N + 15)}{t} * 10^{-9} \tag{7.1}$$

Merania boli realizované pri rôznych konfiguráciách OpenMP - rozmiestnenie, naviazanie na hardvérové časti a počet. Pri meraniach na koprocesore, počet vlákien bol nastavený na hodnoty 60,120,180 a 240. Tieto hodnoty predstavujú konfigurácie 1,2,3 a 4 vlákna na jadro. Keďže koprocesor nie je stavaný na beh nižšieho počtu vlákien, iné konfigurácie počtu vlákien neboli testované.

7.1 Naivná Verzia N-Body

Algoritmus N-Body simulácie je pomerne jednoduchý. Kapitola 5 poskytuje pohľad na princíp fungovania tejto simulácie. Pseudokód naivnej verzie N-Body algoritmu zachytáva úryvok 7.1. Kapitola 5 vysvetľuje matematické vzťahy použité vo funkciách v tomto pseudokóde a tiež obsahuje zjednodušenie týchto vzťahov (vynechanie výpočtu akcelerácie, odstránenie podmienky).

Ďalšia sekcia popisuje základné implementačné detaily.

```
for(step = 0; step < steps; ++step)
{
    for(i = 0; i < N; ++i)
    {
        F = 0;
        for(j = 0; j < N; ++j)
        {
            if(i != j)
                F += calculate_force(particle[i], particle[j]);
        }
        ACC = calculate_acceleration(particle[i], F);
        update_velocity(particle[i], ACC);
        update_position(particle[i]);
    }
}
```

Listing 7.1: Pseudokód N-Body simulácie

7.1.1 Implementácia

Základná sériová verzia sa mierne líši od preusodódu 7.1. Niektoré optimalizácie boli začlenené do programu od začiatku. Konkrétne použitie štruktúry polí na reprezentáciu častíc a

Tabuľka 7.1: Namerané hodnoty pri behu neoptimalizovanej verzie na CPU.

Počet častíc	**Počet iterácií**	**Celkový čas** $[s]$	**Výkon** $[GFLOP/s]$
1024	1000	26,55	0,869
2048	1000	106,41	0,867
4096	1000	426,31	0,866

tiež upravené vzťahy pre výpočet výslednej sily a rýchlosti popísane v kapitole 5.

Každá častica má niekoľko atribútov - hmotnosť, súradnice v priestore a vektor rýchlosti. Tieto atribúty sú uložené v pamäti ako štruktúra polí. Keďže všetky atribúty sú reprezentované rovnakým dátovým typom, namiesto štruktúry bolo použité pole. Teda vo výsledku nie sú dáta uložené vo forme štruktúry polí, ale ako dvojrozmerné pole. Každý atribút je uložený v samostatnom dynamickom poli. Pre zjednodušenie alokácie pamäte bol použitý *std::vector*.

Použitím zjemňujúceho faktora bolo možné vynechať podmienku nerovností častíc pri výpočte celkovej sily pôsobiacej na časticu. V prípade že častica i a j reprezentuje rovnakú časticu, dochádza k deleniu nulou. Zjemňujúci faktor okrem iného zabráni tomuto deleniu nulou.

Ako už bolo spomenuté vyššie, použitím upravených vzťahov na výpočet rýchlosti bolo možné odstrániť jedno násobenie a jedno delenie.

Namerané hodnoty Keďže táto verzia slúži ako základ pri meraní dopadu ostatných optimalizácií, bola skompilované s vypnutými optimalizáciami (prepínač -$O0$). Tabuľka 7.1 zobrazuje namerané hodnoty na CPU. Z nameraných hodnôt je možné vyčítať, že zložitosť tohto prístupu je naozaj $O(N^2)$. Zdrojový kód k tejto časti je možné nájsť v súbore *nbody/naive-openmp-mpi/nbodysimulation.cpp*, konkrétne ide o funkciu *NBodySimulation::runSeq*.

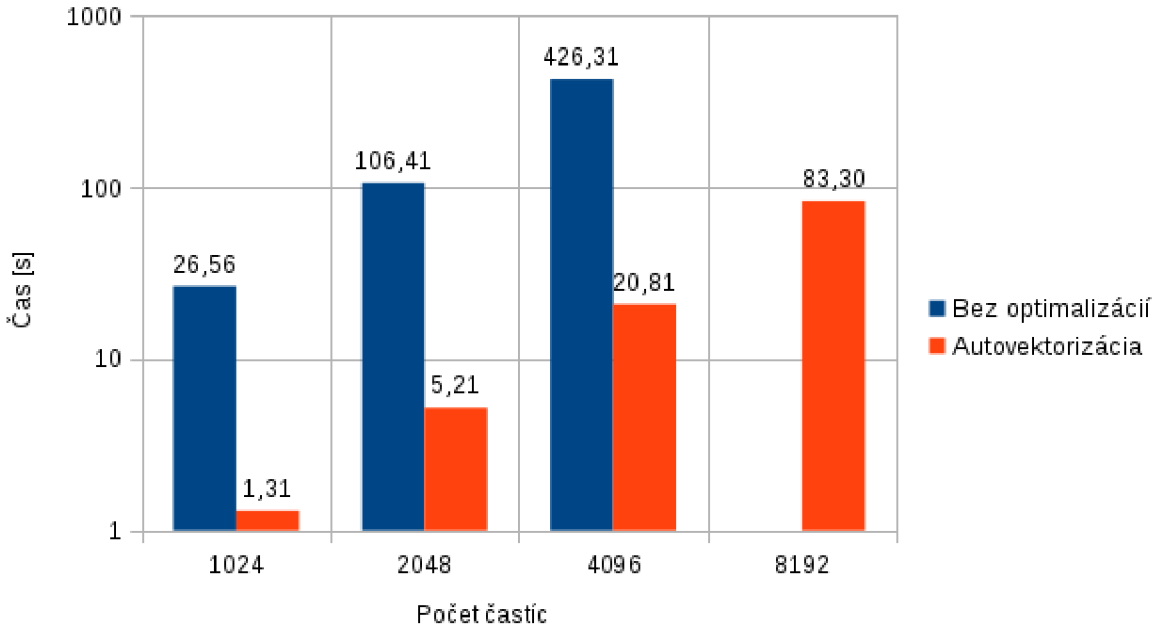
7.1.2 Zarovnanie pamäte, autovektorizácia

Prvá fáza optimalizácie má za cieľ zvýšiť výkon predchádzajúcej verzie použitím vektorizácie. Ako prvá technika, bolo použité zarovnanie pamäte na hodnotu veľkosti najväčších SIMD registrov na danej platforme. Alokácia zarovnanej pamäte je realizovaná pomocou vlastného alokátora *AlignedAllocator*, ktorý na pozadí využíva funkciu *posix_memalign*. Tento alokátor implementuje rozhranie štandardných C++ alokátorov, a preto je ho možné použiť v kombinácií spolu s *std::vector* pre jednoduchú alokáciu polí.

Tabuľka 7.2: Namerané hodnoty pri behu autovektorizovanej verzie na CPU.

Počet častíc	**Počet iterácií**	**Celkový čas** $[s]$	**Výkon** $[GFLOP/s]$
1024	1000	1,31	17,67
2048	1000	5,21	17,73
4096	1000	20,81	17,74
8192	1000	83,30	17,72

Ďalším krokom bolo zapnutie optimalizácií na úrovni kompilátora pomocou prepínača -$O3$. Tento prepínač okrem iného zapne aj automatickú vektorizáciu cyklov. Počas kompilácie boli tiež zapnuté diagnostické hlásenia kompilátora o vektorizácií. Na základe týchto



Obr. 7.1: Dopad autovektorizácie na namerané časy.

vektorizácií je možné zistiť, či daný cyklus bol vektorizovaný. Z diagnostických hlásení vyplývalo, že došlo k vektorizácií najvnútornejšieho cyklu (6. riadok v pseudokóde 7.1), v ktorom sa počítajú čiastkové sily medzi časticami. Z podrobnejšieho skúmania diagnostických hlásení vyplynulo, že kompilátor vygeneroval nezarovnané prístupy do pamäte. Preto bolo potrebné použiť konštrukciu *assume_aligned*. Tabuľka 7.2 zobrazuje namerané hodnoty pri behu autovektorizovanej verzie. Výkon algoritmu vzrástol oproti verzii s vypnutými optimalizáciami 20 násobne. Tento nárast výkonu poukazuje na úroveň optimalizácií, ktoré dokáže súčasný Intel C/C++ kompilátor dosiahnúť v niektorých prípadoch. Graf 7.1 zachytáva dopad autovektorizácie na namerané časy. Zdrojový kód k tejto optimalizácie je možné nájsť v súbore *nbody/naive-openmp-mpi/nbodysimulation.cpp*, konkrétne ide o funkciu *NBodySimulation::runSeq*.

7.1.3 Paralelizácia pomocou OpenMP

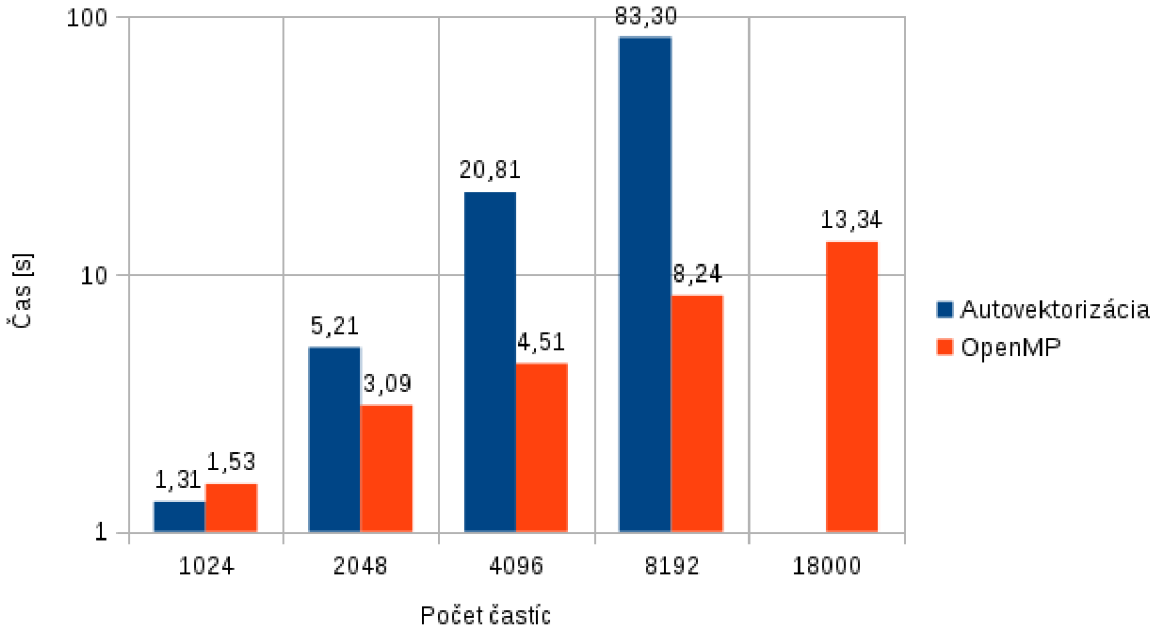
Ďalším krokom na ceste k výkonnému programu je zavedenie paralelizmu na úrovní vlákien. Počas procesu optimalizácie je vhodné použiť nástroje na odhalenie výpočtovo náročných miest pre určenie kandidátov na dodatočnú optimalizáciu a zavedenie paralelizácie. V tomto prípade je hneď zjavné, ktoré miesta sú výpočtovo náročne, keďže hlavný cyklus N-Body simulácie je jednoduchý.

Vytvorenie OpenMP tímu pracovných vlákien so sebou prináša réžiu, preto je vhodné umiestniť celý hlavný cyklus N-Body simulácie umiestniť do OpenMP paralelného regiónu *#pragma omp parallel*. Týmto sa zabráni vytváraniu a rušeniu vlákien v cykle.

Cyklus, ktorý určuje pre ktorú časticu sa aktuálne počíta celková sila je dobrým kandidátom na paralelizáciu na úrovni vlákien. Jeho vnútorný cyklus, v ktorom sa počítajú čiastkové sily medzi časticami bol v predchádzajúcom kroku vektorizovaný. Teda vo výsledku, v každom vlákne bude bežať vektorizovaný kód, čo je veľmi žiadané. V jednotlivých

Tabuľka 7.3: Namerané hodnoty pri behu OpenMP verzie na CPU pri použití 24 vlákien.

Počet častíc	**Počet iterácií**	**Celkový čas** $[s]$	**Výkon** $[GFLOP/s]$
1024	1000	1,53	15,07
2048	1000	3,09	29,83
4096	1000	4,51	81,89
8192	1000	8,35	176,75
18000	1000	13,34	212,32



Obr. 7.2: Zmena časov behu programu pri zavedení paralelizácie na úrovní vlákien.

cykloch sa vykonáva rovnaké množstvo výpočtov, preto bolo použité statické plánovanie (`schedule(static)`).

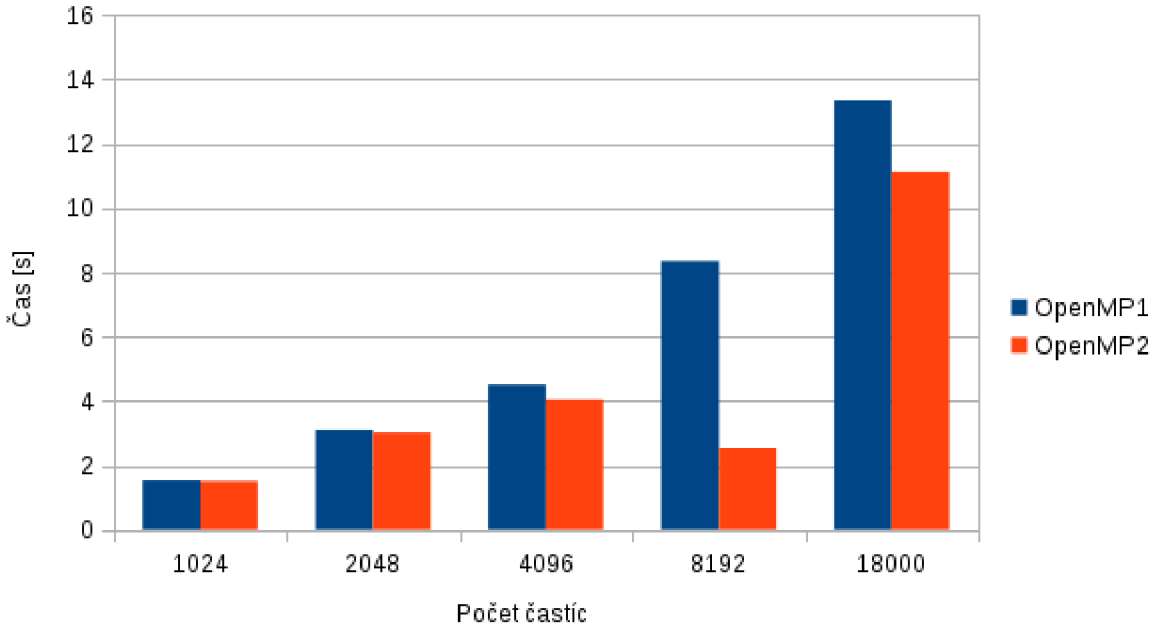
Do algoritmu bola tiež doplnená vektorizovaná redukcia premenných celkovej sily pre danú časticu pomocou OpenMP *#pragma omp simd* ako náhrada za automatickú vektorizáciu z predchádzajúceho kroku. Táto forma bola zvolená z dôvodu lepšieho vyjadrenia zámeru v kóde. Kompilátor si pri automatickej vektorizácií s týmto cyklom poradil veľmi dobre. Kód vygenerovaný kompilátorom pri autovektorizácií sa zhodoval s kódom pri použití implicitnej vektorizácií pomocou OpenMP.

Cieľová platforma pre túto aplikáciu je NUMA systém, kde každý NUMA uzol obsahuje dva fyzické procesory. Preto bola tiež aplikovaná stratégia popísaná v časti 4.7 - NUMA First Touch Policy. Do alokátora *AlignedAllocator* bola pridaná paralelná inicializácia alokovanej pamäte.

Namerané hodnoty z tejto fázy sú v tabuľke 7.3. Pri týchto meraniach bol použitý maximálny počet dostupných jadier na procesore - 24. Z grafu 7.2 je zrejmé, že pri nízkych vstupoch je výkon algoritmu nižší ako pri verzii s jedným vláknom. Je to zrejme spôsobené vysokou réžiou, spôsobenou zavedením viacerých vlákien, v pomere k celkovému času výpočtu. V ostatných prípadoch došlo k nárastu výkonu.

Tabuľka 7.4: Namerané časy pri behu OpenMP verzie na CPU s vektorizovanou aktualizáciou polohy a rýchlostí častíc.

Počet častíc	**Počet iterácií**	**Celkový čas** $[s]$	**Výkon** $[GFLOP/s]$
1024	1000	1,51	15,27
2048	1000	3,02	30,54
4096	1000	4,04	91,419
8192	1000	2,61	565,47
18000	1000	11,12	254,86



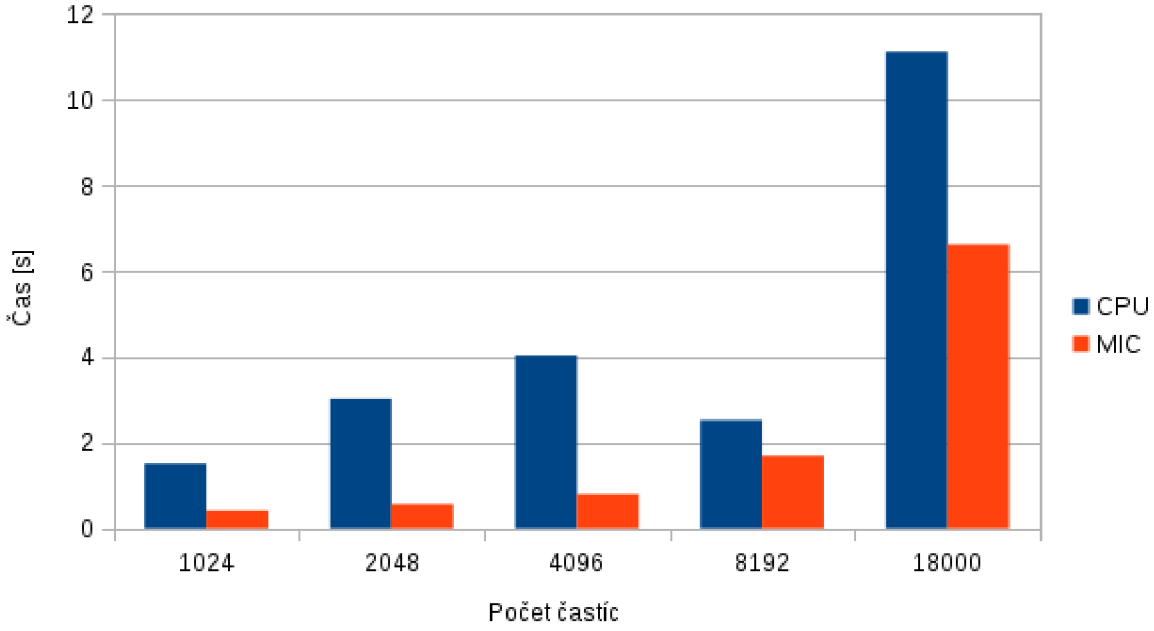
Obr. 7.3: Porovnanie verzie s vektorizovanou aktualizáciou rýchlosti a polohy (OpenMP2) s predchádzajúcou verziou (OpenMP1).

7.1.4 Vektorizácia aktualizácie rýchlosti a polohy

Hlavná časť výpočtu bola vektorizovaná a rozdelená medzi viacero vlákien. Po vypočítaní sily pôsobiacej na časticu je hneď aktualizovaná jej rýchlosť a pozícia. Tieto aktualizácie sú vykonávané pomocou skalárnych inštrukcií, keďže cyklus, v ktorom sa nachádzajú je paralelizovaný na úrovní vlákien.

Presunutím aktualizácie rýchlosti a pozície do osobitného cyklu je možné tieto operácie vektorizovať a zároveň rozdeliť medzi viaceré vlákna. Oddelenie aktualizácie od výpočtu sily však vyžaduje dodatočné úložisko pre sily pôsobiace na častice. Okrem vektorizácie je tu možné zaviesť aj rozdelenie práce medzi viacero vlákien. OpenMP umožňuje zlúčiť pragmy *for* a *simd*. Po oddelení cyklu aktualizácie stačí pred týmto cyklom použiť *#pragma omp for simd*.

Graf na obrázku 7.3 znázorňuje namerané zrýchlenie dosiahnuté pri vektorizovaní aktualizácie polôh a rýchlostí. Pri veľkosti vstupu 8192 častíc môžeme pozorovať takmer štvornásobné zrýchlenie. Toto meranie bolo niekoľkokrát opakované a program bol tiež verifikovaný.



Obr. 7.4: Porovnanie časov behu optimalizovanej verize N-Body na procesore a koprocesore.

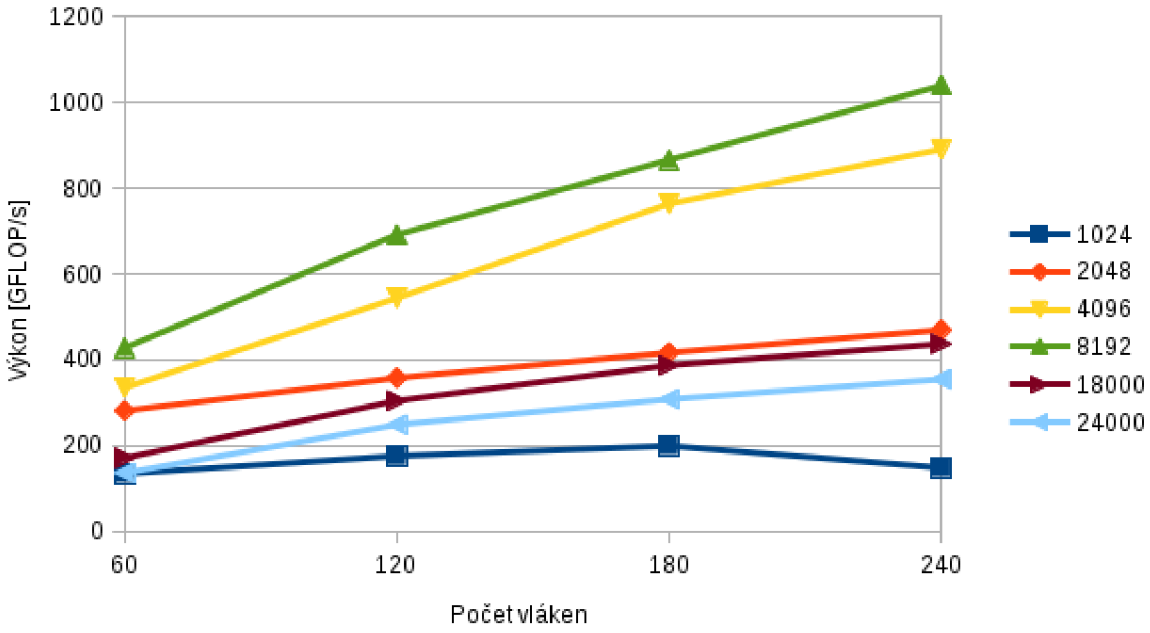
7.1.5 N-Body na Intel Xeon Phi koprocesore

Na základe odporúčaní[9] bol algoritmus najprv optimalizovaný pre Intel Xeon procesor. S využitím Iten VTune nástroja sme zhodnotili že úroveň optimalizácie algoritmu pre procesor je dostatočná. Pri kompilácií programu pre Intel Xeon Phi bol zmenený jeden prepínač, konkrétne *-xhost* bol nahradený za *-mmic*. V zdrojových kódoch neboli vykonané žiadne ďalšie zmeny. Zmena hodnoty pre zarovnanie pamäte prebehla automaticky. Počas kompilácie je detegovaná platforma (resp. podporovaná SIMD inštrukčná sada), pre ktorú je kompilovaný kód a nastavená príslušná hodnota zarovnania. Toto automatické správania je možné prepísať definovaním hodnoty *ALIGNMENT* na príslušnú hodnotu (*-D ALIGNMENT=64*).

Táto verzia programu predstavuje finálnu optimalizovanú verziu N-Body problému pre beh na jednom procesore, respektíve koprocesore. Neboli použité žiadne ďalšie optimalizácie. Čo sa týka optimalizácie zápisov pomocou streaming stores spomenutých v časti 4.5, tie nebolo vhodné použiť. Aktualizácia rýchlosti a pozície vyžaduje čítanie. Pri ukladaní výslednej sily sú hodnoty prepísané, avšak tieto hodnoty sú zapisované v cykle ktorý nie je vektorizovateľný.

Porovnanie priemerných časov behu naivnej N-Body simulácie pri 1000 krokoch na procesore a koprocesore je na obrázku 7.4. Naivná verzia N-Body simulácie je dobre paralelizovateľná a vektorizovateľná, čo sa prejavilo na výsledných časoch. Ako je zmienené vyššie, pri "portovaní"progrmu na koprocesor z procesora, boli vykonané len dve malé zmeny. Jedna v zdrojovom kóde a jedna v kompilačnom skripte.

Najvyšší výkon bol dosiahnutý pri veľkosti vstupu 8192 a 4096 častíc. Pri väčších vstupoch výkon vzhľadom k počtu vlákien rastie pomalšie, čo je zapríčinené pravdepodobne nedostatkom cache pamäte. Pri nízkych vstupoch je výkon negatívne ovplyvnený nízkym množstvom práce. Škálovanie výkonu na koprocesore s rasúcim počtom vlákien je zobrazené



Obr. 7.5: Škálovanie naivnej verzie N-Body na koprocesore.

v grafe 7.5. V prípade 8192 častíc, došlo oproti jednovláknovej verzii s autovektorizáciou na procesore až k 50-násobnému zrýchleniu.

7.1.6 Distribuovanie výpočtu

Algoritmus optimalizovaný v predchádzajúcich častiach bol na záver rozšírený tak, aby pracoval v distribuovanom prostredí. Problém bol rozdistribuovaný medzi procesy rovnakým dielom. Na začiatku každý proces určí, ktorú časť častíc bude spracovávať. Keďže pre výpočet celkovej sily pre každú jednu časticu sú potrebné informácie o polohe všetkých častíc, na začiatku sú všetky častice odoslané každému procesu. Po každej iterácií, po vykonaní aktualizácie pozície a rýchlosti, je potrebné synchronizovať aktualizované polohy častíc, ktoré sú potrebné pre výpočet ďalšej iterácie. Na tento prenos pozícií je použitá kolektívna funkcia *Allgather*. Procesy si medzi sebou navzájom vymenia ich nové, lokálne vypočítané, pozície. Rýchlosti nie je potrebné synchronizovať.

Kapitola 8 sa venuje porovnávaniu behu finálnej optimalizovanej verzie v distribuovanom prostredí. Otestovaných bude niekoľko rôznych konfigurácií.

7.2 Barnes-Hut Verzia N-Body

Barnes-Hut algoritmus bol predstavený v časti 5.1. Táto kapitola sa zaoberá implementáciou a optimalizáciou tohto problému. Prvá časť tejto kapitoly opisuje použité prístupy k implementácií jednotlivých fáz algoritmu a tiež analyzuje MPI komunikáciu potrebnú pre beh riešenia v distribuovanom prostredí v rámci Salomon klastra. V ďalšej časti sú popísane použité optimalizácie s meraniami ich dopadu na celkový výkon algoritmu.

```
for(step = 0; step < steps; ++step)
{
    buildOctree(octree,particles);
    sumarizeSubtrees(octree);

    computeForces(octree,particles);

    advance(particles,timestep);
}
```

Listing 7.2: Pseudokód N-Body Barnes-Hut simulácie

Pseudokód 7.2 zachytáva hlavné časti Barnes-Hut algoritmu. Zo pseudokódu je zjavné, že algoritmus pracuje v troch fázach, zo štyroch, s oktálovým stromom. Pri návrhu dátových štruktúr a algoritmov jednotlivých fáz bolo potrebné brať do úvahy, že cieľovou platformou je výpočtový klaster. Implementácia čerpá inšpiráciu z článku "24,77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs."[3].

7.2.1 Implementácia

Barnes-Hut N-Body simulácia pozostáva so štyroch hlavných krokov: zostavenie okátlového stromu, výpočet ťažísk a hmotností v rámci uzlov stromu, výpočet síl prechodom cez strom a na záver aktualizácia polohy a rýchlosti častíc.

Reprezentácia dát

Keďže algoritmus vyžaduje oktálový strom, bolo potrebné zvoliť spôsob reprezentácie tohto stromu v pamäti. Rovnako ako v prípade naivnej verzie sme zvolili štruktúru polí namiesto zvyčajného prístupu s využitím štruktúr. Reprezentácia stromu bola dekomponovaná na jednotlivé zložky. Každý uzol v tomto strome obsahuje nasledujúce atribúty: ťažisko, hmotnosť a počet častíc nachádzajúcich sa v podstromoch tohto uzlu. Každá úroveň stromu má vyhradené samostatné pole, ktoré ďalej obsahuje polia atribútov. Kód 7.3 zobrazuje deklaráciu týchto polí a príklad, ako pristupovať k jednotlivým atribútom. Prvý index udáva hĺbku, v ktorej sa nachádza daný uzol, druhý predstavuje atribút (hmotnosť alebo jednotlivá súradnice ťažiska). Posledný index vyjadruje o ktorý konkrétny uzol sa jedná. Poradie týchto indexov môže pôsobiť nelogicky (najprv sa udáva atribút a až potom o ktorý uzol sa jedná). Vyplýva to z použitia štruktúry polí.

```
std::vector<std::vector<AlignedVector<float>>> nodes;
std::vector<AlignedVector<uint32_t>> nodes_particle_counts;

mass = nodes[3][MASS][511];
node_particles = nodes_particle_counts[3][511];
```

Listing 7.3: Deklarácia uzlov oktálového stromu a príklad získania atribútu uzlu

Ide o úplný oktálový strom. Každý uzol má osem potomkov. Potomkovia uzlu sa nachádzajú v nasledujúcom poli voči rodičovi. Index prvého potomka p_0 je možné určiť pomocou vzťahu $p_0 = 8 * r$, kde r je index rodiča. Indexy ostaných potomkov sú $p_0 + 1$ až $p_0 + 7$.

Jednotlivé častice sú uložené oddelene, nezávislé na strome. Reprezentácia častíc je rovnaká ako v prípade naivnej verzie. Popis je možné nájsť v časti 7.1.1. Prepojenie medzi oktálovým stromom a úložiskom častíc je znázornenie na obrázku 7.6. Listové uzly majú ďalší atribút - index do úložiska častíc. Pomocou tohto indexu a počtu častíc v listovom uzle je možné iterovať cez všetky častice prislúchajúcej tomuto uzlu.

Všetky relevantné dátové štruktúry sú zaobalené v triede *Octree*. Definíciu tejto triedy je možné nájsť v *súbore nbody/barnes_hut/octree.h*.

Zostavenie oktáloveho stromu

Pre beh algoritmu je nevyhnutné zostrojiť oktálový strom, ktorý zoskupí častice podľa ich polohy. Klasický prístup pri vytváraní tohto stromu založený na postupnom vkladaní častíc do stromu nie je príliš vhodný pre paralelné prostredie. Tento prístup vyžaduje značnú synchronizáciu medzi vláknami. Alternatívnou metódou je použiť krivky vyplňujúce priestor (angl. Space Filling Curves, ďalej len SFC). Táto metóda je z časti založená na metóde použitej v článku Ä sparse octree gravitational N-body code that runs entirely on the GPU processor"[4]. Prvým rozdielom je, že sme použili úplný strom namiesto riedkeho. Ďalšou úpravou je smer zostavovania stromu. V našom prípade sme postupovali zdola hore.

Proces zostavovania stromu pozostáva z niekoľkých krokov:

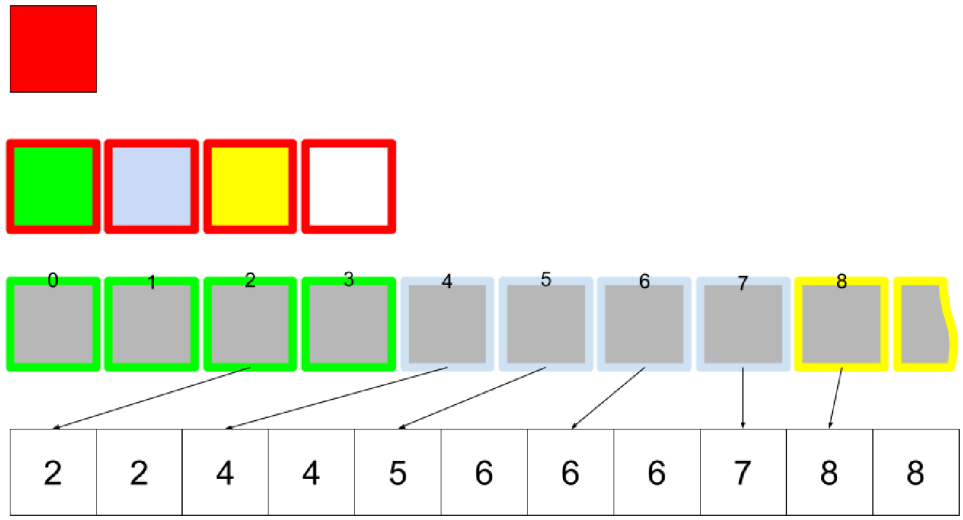
- Výpočet SFC indexu pre všetky častice
- Utriedenie častíc podľa SFC indexu
- Prepojenie stromu s úložiskom častíc
- výpočet atribútov uzlov stromu

SFC majú zaujímavú vlastnosť - bodom v N-rozmernom priestore priradzujú bod v jednorozmernom priestore. Tento bod budeme ďalej nazývať SFC index. Existuje niekoľko typov SFC kriviek. Pre nás zaujímavými sú hlavne krivky ako Mortonova Z-krivka (obrázok 7.7 alebo Hilbertova krivka. Ak sú SFC indexy určené týmito krivkami blízko seba, potom aj zdrojové body sú blízko seba (až na niekoľko krajných prípadov, napríklad pre body [4,4] a [3,3] v 2d priestore o rozmeroch 8x8, kde ich príslušné SFC indexy sú 15 a 50).

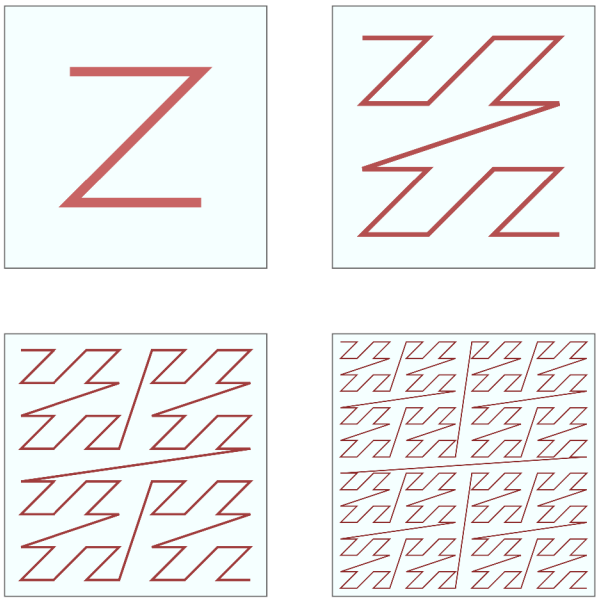
Pri implementácií sme použili Mortonovu Z-krivku kvôli jej výpočtovej jednoduchosti. Pre namapovanie hodnoty z 3D do 1D stačí niekoľko binárnych operácií. Implementácia výpočtu SFC indexu tejto krivky sa nachádza v súbore *nbody/barnes_hut/octree.cpp* vo funkcii *get_key_morton*.

SFC indexy boli využité na rozdelenie všetkých častíc do skupín. Každá takáto skupina častíc bola priradená jednému listu v strome, pričom priradenie bolo nasledovné: listovému uzlu s indexom i je priradený SFC index i. Rekurzívnosť SFC zabezpečuje, že po usporiadaní do skupín a priradení týchto skupín do listov stromu, dosiahneme priestorové rozdelenie častíc podľa pravidiel stavby okálového stromu (s variáciou že body sú umiestnené v listoch v skupinách).

Pozícia častíc nie je vo vhodnej forme pre výpočet SFC indexov. Najprv je potrebné pozície transformovať a zoskupiť do skupín. Transformácie vyžadujú znalosť obalovej kocky okolo všetkých častíc a počet listov v strome. Na základe týchto informácií je aplikovaná transformácia posunutia na všetky častice a tiež transformácia zmeny mierky. Výsledkom sú súradnice polohy častíc v intervale $\left\langle 0, \sqrt[3]{N} \right)$, kde N je počet listov v strome. Odstránením desatinnej časti dostaneme hodnotu, ktorá je použitá na výpočet SFC indexu danej častice.



Obr. 7.6: Schéma reprezentácie stromu (obrázok znázorňuje quadtree). Farebné štvorce predstavujú uzly stromu. Uzly na rovnakej úrovni sú uložené v pamäti lineárne. Rovnaká farba výplne a ohraničenia reprezentuje vzťah rodič-potomok (výplň-ohraničenie). Listy stromu sú vyplnené sivou farbou a ich čísla predstavujú index uzlu v poli. Posledná úroveň (biela výplň) predstavuje úložisko častíc a ich číslo značí SFC index. Šípky označujú prepojenie medzi stromom a úložiskom stromu. Listový uzol s indexom i odkazuje na prvý výskyt SFC indexu i v utriedenom poli častíc podľa SFC indexov.



Obr. 7.7: Príklad krivky vyplňujúcej priestor (SFC) zvanej Mortonova Z-krivka a jej štyri iterácie. Zdroj: https://en.wikipedia.org/wiki/Z-order_curve

Implementácie tejto metódy sa nachádza vo funkcii *Octree::computeLocalSFCKeys* v súbore *súbore nbody/barnes_hut/octree.cpp*.

Ďalšou fázou stavby oktálového stromu je utriedenie častíc podľa SFC indexov. Častice sú ale reprezentované štruktúrou polí, preto tu nie je možné použiť triediaci algoritmus priamo kvôli zachovaniu správneho poradia atribútov. Problém je možné vyriešiť vytvorením dočasného poľa, ktoré sa naplní postupnosťou od 0 až počet častíc−1 a jeho následným triedením použitím špeciálneho operátora. Každý prvok v tomto pomocnom poli reprezentuje zástupcu jednej častice a index tohto prvku značí poradie tejto častice. Iniciálny stav pomocného poľa reprezentuje aktuálne poradie častíc.

Toto pomocné pole je následne utriedené. Neporovnávajú sa hodnoty priamo v tomto poli. Pri porovnávaní hodnoty i a j je namiesto porovnania týchto dvoch hodnôt porovnaný SFC index častice i a SFC index častice j. Výsledné utriedené pomocné pole vyjadruje, ktorá častica patrí na ktorý index. Hodnota c v pomocnom poli na indexe i znamená, že častica c sa v utriedenej postupnosti častíc nachádza na pozícií i. Zostáva už len preusporiadať všetky atribúty častí na základe pomocného poľa, výsledkom čoho je utriedená postupnosť častíc podľa SFC indexu. Triedenie je implementované vo funkcii *Octree::buildLocalSortedSFCIndices*.

Ostáva už len vyplniť uzly oktálového stromu. Najprv sú vypočítané počty častíc v jednotlivých skupinách (častíc s rovnakým SFC indexom). Tieto hodnoty sú vložené do listov stromu a následne sú dopočítané počty častíc v ostatných uzloch zdola nahor. Ďalej je vytvorené spojenie medzi listami stromu a úložiskom častíc. Výpočet vychádza z počtu častíc v jednotlivých listoch. Nakoniec sú do stromu doplnené súčty hmotností a ťažiská v jednotlivých uzloch. Tiež zdola nahor. O túto funkcionalitu sa stará funkcia *Octree::buildLocalOctree*. Ťažisko je vypočítané podľa vzťahu 7.2, kde c je ťažisko, $\mathbf{r}_i$ je pozícia častice i a m_i je hmotnosť častice i.

$$\mathbf{c} = \frac{\sum_{i=1}^{N} m_i \mathbf{r}_i}{\sum_{i=0}^{N} m_i} \tag{7.2}$$

Výpočet síl a aktualizácie častíc

Výpočet celkovej sily spočíva v prechode stromom, raz pre každú časticu. V každom kroku je vyhodnotené, či algoritmus bude pokračovať v prehľadávaní daného podstromu, alebo pre výpočet sily použije ťažisko. Popis kritéria pre vynechanie podstromu a príslušné vzťahy sú v kapitole 5.1. Výsledná sila každej častice je uložená do pomocného poľa. Po skončení výpočtu síl sú všetky častice naraz aktualizované.

Implementované boli dva rôzne prístupy k prehľadávaniu stromu. Prehľadávanie do hĺbky a prehľadávanie do šírky. Prehľadávanie do hĺbky je implementované pomocou rekurzie. Podmienka ukončenia rekurzie je rozšírená o podmienku Barnes-Hut algoritmu.

Prehľadávanie do šírky využíva pomocné zoznamy, pre každú úroveň jeden. Veľkosť tohto zoznamu je rovná počtu uzlov v jednotlivých úrovniach. V týchto poliach je uložený príznak, ktorý reprezentuje, či potomci daného uzla budú ďalej prehľadávané. Pri prehľadávaní je vynechané prehľadávanie uzlov v prvých dvoch úrovniach, keďže tieto uzly nie sú nikdy preskočené.

Úpravy algoritmu pre beh v distribuovanom prostredí

Program bol od začiatku vyvíjaný ako MPI aplikácia. Predchádzajúca časť opisovala algoritmus pre beh aplikácie v rámci jedného procesu. Iniciálna distribuovaná verzia využíva blokujúcu komunikáciu.

Distribuovaná verzia riešenia problému spočíva v kolektívnom vytvorením globálneho oktálového stromu, ktorý je synchronizovaný medzi všetkými procesmi. Každý proces potom pomocou tohto stromu spracuje časť častíc.

Počas inicializácie sú častice rozdistribuované medzi všetky procesy. Každému procesu je priradené časť častíc, ktorú bude ďalej spracovávať.

Pri výpočte SFC indexov je nevyhnutná informácia o obalovej kocke všetkých častíc. Každý proces vypočíta obalový kváder pre častice, ktoré mu boli pridelené. Redukciou lokálnych obalových kvádrov je vypočítaný globálny obalový kváder. Keďže každý proces potrebuje informáciu o globálnej obalovej kocke pre výpočet SFC indexov, bola použitá redukcia do všetkých procesov - *MPI_Allreduce*. Táto redukcia bola použitá dvakrát - na nájdene minimálnych a maximálnych súradníc. Každý proces si následne vypočíta stred a dĺžku strany obalovej kocky (maximálna dĺžka zo všetkých strán obalového kvádra).

Po výpočte SFC indexov nasleduje triedenie. Lokálne triedenie SFC indexov je ale nedostatočné. Algoritmus vyžaduje, aby bolo globálne pole častíc usporiadané podľa SFC indexov, preto je nutné použiť distribuovaný triediaci algoritmus. Zvolili sme Merge-splitting sort algoritmus, ktorý je optimálny pre $p \leq logN$, kde p kde počet procesov a N je veľkosť vstupného vektora. Každý proces pred začatím distribuovaného triedenia utriedi pole pomocných indexov podľa SFC indexov (princíp popísaný v predchádzajúcej časti). V každej iterácií prebehne niekoľko komunikácií medzi procesmi. Každý proces s párnym ID (MPI rank) si najprv vymení svoju postupnosť s pravým susedom. Procesy tieto dve postupnosti zlúčia do jednej (pomocou *std::merge* s využitím modifikovaného porovnávacieho operátora, ako pri triedení poľa pomocných indexov podľa SFX indexov) a nahradia svoju postupnosť polovicou tejto zlúčenej. Zlúčenie dvoch utriedených postupností je rýchlejšie ako použitie triediaceho algoritmu. Každý proces nahradí svoju postupnosť jednou polovicou zo zlúčenej

postupnosti (proces s menším ID si stále zvolí prvú polovicu). V rámci jednej iterácie je vykonaný ešte jeden takýto krok. Tentokrát ale párny proces komunikuje s ľavým susedom. Pre utriedenie celého vstupu, algoritmus vyžaduje $\lceil p/2 \rceil$ takýchto iterácií, kde p je počet procesov. Po skončení triedenia, každý proces má k dispozícií časť utriedenej postupnosti pomocných indexov.

Rovnako ako pri použití jedného procesu, nasleduje preusporiadanie častíc. Každý proces si z globálneho poľa častíc prekopíruje do lokálnej pamäte častice podľa pomocného poľa indexov. Zozbieraním všetkých lokálnych častíc do globálneho poľa dostaneme utriedené častice v globálnom poli podľa SFC indexov.

Fáza zostrojenia stromu bola obohatená o niekoľko kolektívnych operácií. Pri zostavovaní lokálneho stromu, ak listový uzol neobsahuje platný index do úložiska častíc, je jeho hodnota nastavená na maximálnu možnú hodnotu. Táto zmena je dôležitá pri formovaní globálneho stromu.

Zostavenie globálneho stromu z lokálnych je realizované pomocou niekoľkých redukcií nad lokálnymi stromami. Pre dodržania vzťahu 7.2 pre výpočet ťažiska, je na začiatku tohto procesu vykonané prenásobenie pozícií ťažiska v každom uzle hmotnosťou v tomto uzle. Pomocou redukcie do každého procesu (*MPI_Allreduce*) je vypočítaná globálna hmotnosť, počet častíc a pozícia ťažiska v každom uzle (redukcia súčtu) a indexy v listových uzloch do úložiska častíc (redukcia minima). Pri indexoch do úložiska častíc je možné použiť redukciu minima z dôvodu, že neplatné indexy majú hodnotu rovnú maximálnej hodnote daného typu premennej a validné hodnoty sú menšie. Ak rovnaký listový uzol v rôznych lokálnych stromoch obsahuje index do globálneho úložiska častíc, je použitá najmenšia hodnota. Na záver sú pozície vo všetkých uzloch vydelené celkovou hmotnosťou v danom uzle, čím získame výsledné ťažisko.

Aktualizáciu pozícií a rýchlostí vykonáva každý proces nezávisle na ostatných. Po dokončení aktualizácie sú atribúty častíc zosynchronizované medzi všetkými procesmi pomocou *MPI_Allgather*.

Funkcie implementujúce túto funkcionalitu sa nachádzajú v súbore *octree.cpp* v zložke *nbody/barnes_hut/*.

Verifikácia

Podobne ako v prípade naivnej N-Body verzie, po každej fáze vývoja, algoritmus bol verifikovaný. Kontrolný súčet bol implementovaný rovnako ako pri naivnej verzii - súčet absolútnych hodnôt všetkých atribútov. Barnes-Hut algoritmus aproximuje riešenie N-Body problému. Chyba algoritmu do značnej miery závisí na parametri θ. Pri použití $\theta = 0$ algoritmus degraduje na $O(N^2)$. Algoritmus bol verifikovaný pre $\theta = 0$ a tiež pre $\theta = 0,5$. Ako referencia slúžila sekvenčná autovektorizovaná verzia naivného N-Body riešenia. Maximálna hĺbka stromu bola v tomto prípade nastavená na 3, čo predstavuje 64 uzlov na najnižšej úrovni.

Meranie behu základnej verzie

Táto verzia bez aplikovaných optimalizácií slúži ako základ pre meranie dopadu optimalizácií. Program bol skompilovaný s vypnutými optimalizáciami na strane kompilátora. V tejto verzii neisu použité žiadne OpenMP konštrukcie. Testovanie prebiehalo pri behu jedného procesu. V tabuľke 7.5 sa nachádzajú namerané hodnoty pre dva vstupy a niekoľko hodnôt parametra θ. Z tabuľy je zrejmé, že parameter θ, a tiež maximálna hĺbka stromu, do veľkej

Tabuľka 7.5: Namerané hodnoty pri behu neoptimalizovanej Barnes-Hut verzie na CPU s využitím rekurzie na prechod stromom.

Počet častíc	θ	**Počet iterácií**	**Maximálna hĺbka stromu**	**Vynechaných interakcií [%]**	**Čas [s]**
1024	0	1000	3	0	79,90
1024	0,5	1000	3	53,92	38,81
1024	0,5	1000	4	76,82	30,69
1024	0,75	1000	3	80,32	17,98
1024	0,75	1000	4	88,90	15,09
2048	0	1000	3	0	305,43
2048	0,5	1000	3	49,45	158,27
2048	0,5	1000	4	79,67	88,23
2048	0,75	1000	3	63,35	114,26
2048	0,75	1000	4	89,32	45,86

miery ovplyvňuje celkový čas. Pre maximálnu hĺbku na úrovni 3 je počet listových uzlov stromu rovný 64, v prípade maximálnej hĺbky 4 ide o hodnotu 512.

7.2.2 Optimalizácia

Optimalizácia programu prebiehala v niekoľkých fázach. Najprv bola program optimalizovaný pre beh programu v rámci jedného procesu. Optimalizácie pre koprocesor boli pridané po dosiahnutí rozumnej úrovne optimalizácie na CPU. Zarovnanie pamäte a NUMA stratégia prvého dotyku boli zahrnuté už v základnej verzii.

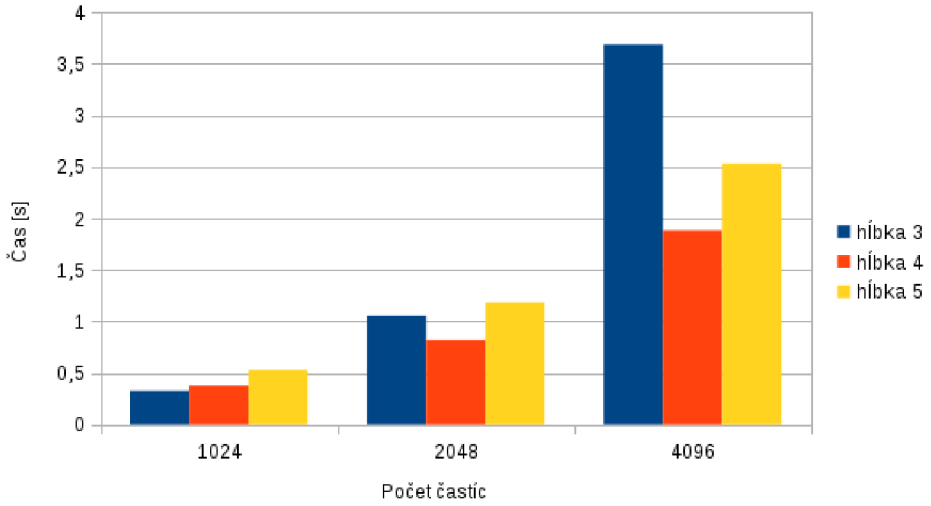
Autovektorizácia

V prvej fáze optimalizácie bola použitá autovektorizácia spolu s ďalším automatickými optimalizáciami. Po skompilovaní programu s pridaním parametra na zapnutie optimalizácií (*-O3*) boli analyzované diagnostické hlásenia kompilátora o použitých optimalizáciách. Následne bol program analyzovaný pomocou nástroja Intel Advisor.

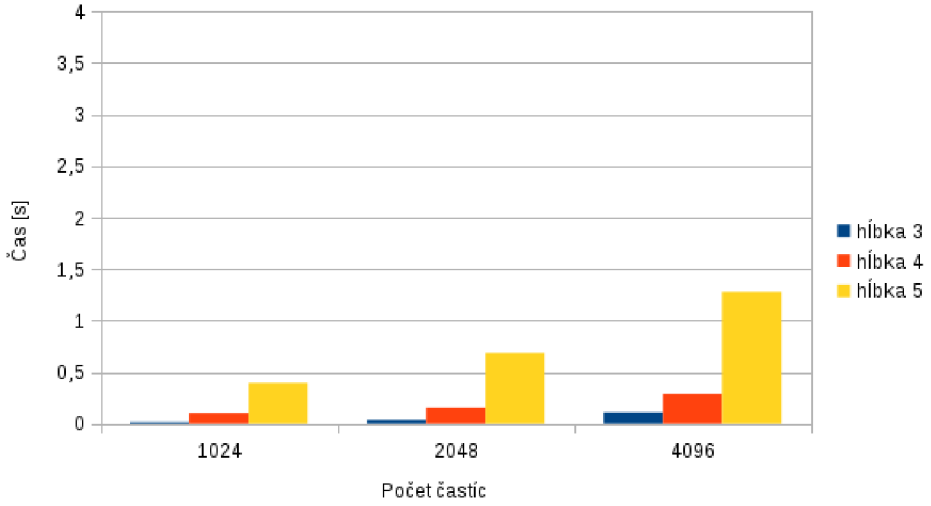
Na základe týchto údajov boli doplnené chýbajúce *ASSUME_ALIGNED* direktívy v prípadoch, keď kompilátor sám nedokázal detegovať úroveň zarovnania pri použití zložitejších dátových typov Pri niektorých cykloch, ktoré obsahovali v podmienke cyklu členské premenné triedy, kompilátor nepoužil autovektorizáciu z dôvodu, že nedokázal odhadnúť počet iterácií. Tieto cykly, ak to logika cyklu dovoľovala, boli vektorizované manuálne pomocou OpenMP. Pri výpočte SFC indexov častíc bolo potrebné manuálne vektorizovať cyklus, špecifikovať dĺžku SIMD pomocou klauzuly *safelen* a tiež deklarovať funkcie na výpočet SFC indexu pomocou OpenMP direktívy *declare simd*.

Zavedenie paralelizmu na úrovní vlákien

Autovektorizovaný program bol následne analyzovaný pomocou nástroja Intel VTune Amplifier. Na základe tejto analýzy bolo zistené, že program strávi väčšinu času výpočtom síl pôsobiacich na častice. Časy vykonávania ostatných fáz N-Body simulácie boli zanedbateľné v porovnaní s výpočtom síl. Preto sme sa zamerali len na optimalizáciu tejto fázy. V prípade dosiahnutia lepších výsledkov, budú následne optimalizované aj ostatné fázy algo-



Obr. 7.8: Celkový čas 1 kroku N-Body simulácie pri použití prehľadávania do hĺbky na výpočet celkovej sily pôsobiacej na časticu. Parametre: 24 vlákien, $\theta = 0,5$



Obr. 7.9: Celkový čas 1 kroku N-Body simulácie pri použití prehľadávania do šírky na výpočet celkovej sily pôsobiacej na časticu. Parametre: 24 vlákien, $\theta = 0,5$

Tabuľka 7.6: Profil algoritmu. Percentuálne vyjadrenie doby trvania fázy algoritmu vo celkovému času. Parametre: $\theta = 0,5$, max. hĺbka stromu = 4.

Počet častíc	**Zostavenie stromu**	**Výpočet síl**	**Aktualizácia častíc**
1024	5,7 %	93,8 %	0,5 %
2048	6,1 %	93,4 %	0,5 %
4096	6,0 %	93,4 %	0,6 %
8192	5,4 %	94,1 %	0,5 %
18000	4,9 %	94,6 %	0,5 %

ritmu. Profil algoritmu s prehľadávaním do šírky je zobrazený v tabuľke 7.6. Pre ostatné vstupné parametre bol profil približne rovnaký.

Rozdelenie práce medzi jednotlivé vlákna je na úrovni výpočtu sily jednej častice. Každé vlákno má pridelenú časť častíc. Pre každú časticu následne postupne prechádza stromom a akumuluje sily pôsobiace na časticu.

Obrázky 7.8 a 7.9 zachytávajú priemerné namerané časy behu 1 kroku N-Body Barnes-Hut simulácie. Graf 7.8 obsahuje prehľadávanie stromu do hĺbky pomocou rekurzívneho volania funkcie. Prehľadávanie do šírky je na obrázku 7.9. Počet vlákien počas týchto meraní bol nastavený na 24. Parameter θ bol nastavený na $0,5$.

Barnes-Hut N-Body na Intel Xeon Phi koprocesore

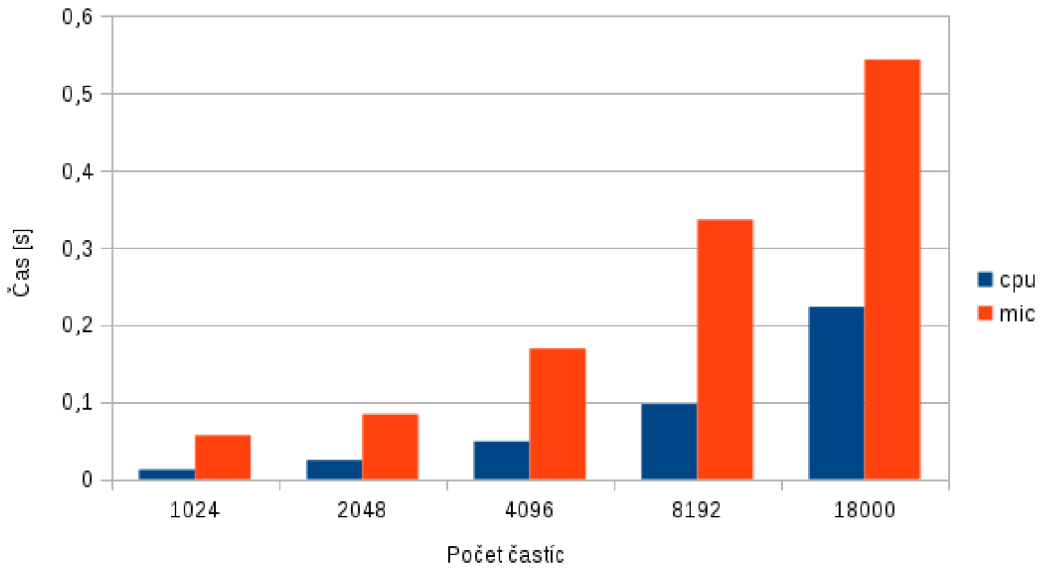
Po aplikácií paralelizmu na úrovni vlákien sa nám nepodarilo zaviesť ďalší paralelizmus pri výpočte síl pôsobiacich na častice. Algoritmus prehľadávania do hĺbky bol doplnený a konštrukcie *ASSUME_ALIGNED* a prístupy k dátam v *std::vector*, respektíve v *AlignedVector* boli upravené na priamy prístup cez ukazovatele na polia. Tieto zmeny však nepriniesli výrazný pokrok v optimalizácií algoritmu.

Ostatné fázy algoritmu neboli už ďalej optimalizované, keďže prechod stromom stále zaberal väčšinu času celkového behu programu. Verzia s prehľadávaním do hĺbky nebola ďalej podrobné testovaná. Zamerali sme sa na verziu s prehľadávaním do šírky, keďže táto verzia dosiahla pri behu na CPU výrazne lepšie výsledky.

Graf na obrázku 7.10 zobrazuje porovnanie priemerného vykonanie 1 kroku simulácie na procesore a koprocesore. Meranie prebiehalo pri použití maximálneho počtu vlákien. Z grafu možno určiť, že procesor dosiahol lepšie výsledky v porovnaní s koprocesorom. Hlavnou príčinou ja nízka úroveň vektorizácie v tomto algoritme a teda neefektívne využitie dostupných zdrojov koprocesora.

Zmenu času s rastúcim počtom vlákien pre rôzne počty častíc na vstupe zachytáva graf na obrázku 7.11. Jednotlivé podgrafy reprezentujú škálovanie pre rôzne úrovne maximálnej hĺbky oktálového stromu (maximálne hĺbky zľava doprava - 4, 5 a 6).

So zvyšujúcou sa maximálnou hĺbkou stromu môžeme pozorovať tiež zvýšenie času na vykonanie jednej iterácie. Maximálna hĺbka priamo ovplyvňuje počet častíc na uzol. Pri vstupe 18000 častíc a rovnomernom rozložení častíc v priestore, pri maximálnej hĺbke 4 pripadá na jeden uzol 35 častíc, pri hĺbke 5 to je približne 5 častíc na uzol a pri hĺbke 6 polovica listov obsahuje 1 časticu. Algoritmus pracuje lepšie pri väčšom počte častíc na uzol.



Obr. 7.10: Porovnanie priemerného času jedného kroku Barnes-Hut N-Body simulácie na procesore a koprocesore.

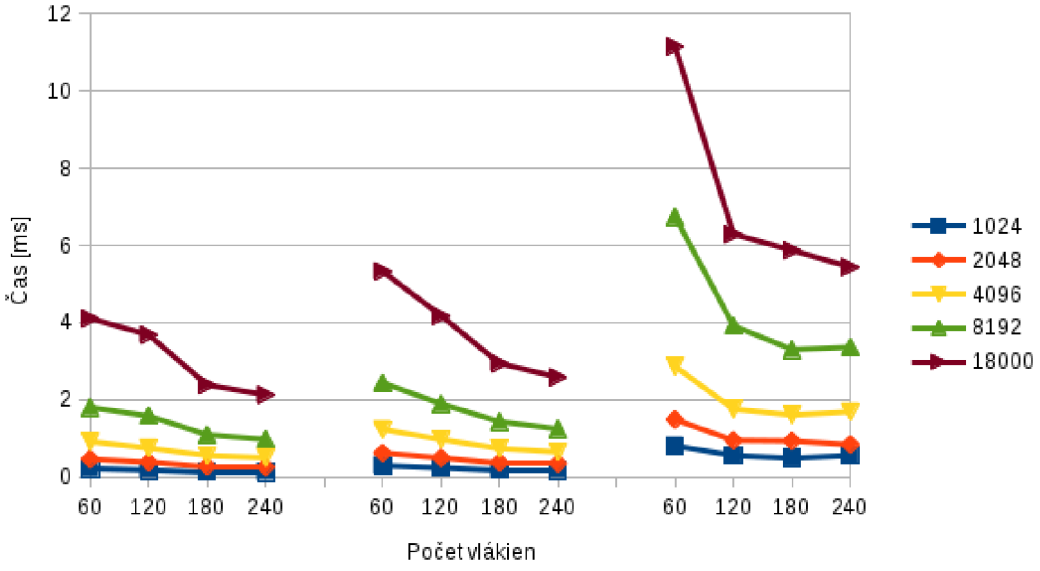
Optimalizácia MPI komunikácie

Podkapitola 7.2.1 opisuje princíp rozloženia výpočtu Barnes-Hut N-Body simulácie medzi viacero uzlov. Ako už bolo spomenuté, algoritmus bol od začiatku vyvíjaný pre distribuované prostredie. Komunikácia medzi procesmi bola implementovaná ešte pred optimalizáciou tohto programu pre behu v rámci jedného procesu. Vo všetkých prípadoch bola použitá blokujúca komunikácia. V niektorých prípadoch je možné získať zrýchlenie použitím neblokujúcej komunikácie a prekrytím komunikácie s výpočtom, ako je popísané v časti 4.8.1.

Prvým krokom k prekrytí výpočtu s komunikáciou je analýza dátových závislosti medzi jednotlivými funkciami algoritmu. Je potrebné odhaliť miesta v programe, kde je možné iniciovať prenos dát a tiež miesta, ktoré vyžadujú aby dané dáta boli dostupné. Tento algoritmus využíva komunikáciu medzi procesmi na niekoľkých miestach. Komunikáciu počas inicializácie nebudeme uvažovať. Nasledujúce metódy triedy *Octree* vyžadujú komunikáciu:

- `computeGlobalAABB` - *MPI_Allreduce*
- `mpiAllGatherSFCKeys` - *MPI_Allgather*
- `mpiSortSFCIndices` - *MPI_Sendrecv*
- `mpiAllGatherParticles` - *MPI_Allgather*
- `mpiBuildGlobalOctree` - *MPI_Allreduce*
- `mpiAllGatherPosVel` - *MPI_Allgather*

Neblokujúcu komunikáciu je vhodné použiť v prípadoch, ak medzi miestom iniciácie komunikácie a miestom kde sú dané dáta vyžadované je dostatočná vzdialenosť, respektíve medzi týmito miestami sa nachádza ďalší výpočet, ktorým môžeme prekryť komunikáciu. Kandidátmi pre zavedenie neblokujúcej komunikácie sú nasledujúce metódy triedy *Octree*:



Obr. 7.11: Škálovanie behu simulácie na koprocesore. Jednotlivé skupiny v grafe predstavujú merania s rôznou maximálnou hĺbkou oktálového stromu.

- `mpiAllGatherSFCKeys` - Po vypočítaní SFC indexov lokálnych častíc, každý proces potrebuje informácie o SFC indexoch všetkých častíc pre utriedenie častíc podľa SFC indexu na globálnej úrovni. Medzi výpočtom lokálnych SFC indexov a globálnym triedením podľa SFC indexov sa nachádza triedenie pomocného poľa podľa SFC indexov.

- `mpiAllGatherParticles` - Táto metóda aktualizuje globálne úložisko častíc na základe preusporiadaného lokálneho úložiska častíc. Prístup ku globálnemu úložisku častíc je vyžadovaný až pri prechádzaní stromu pri výpočte síl. Medzi touto metódou a výpočtom síl sa nachádza stavba lokálneho stromu a tiež stavba globálneho stromu MPI redukciou.

- `mpiAllGatherPosVel` - Táto metóda zosynchronizuje častice v globálnom úložisku každého procesu po ich aktualizácií na konci každého kroku simulácie. Prístup ku globálnemu úložisku častíc je vyžadovaný až v ďalšom kroku simulácie. Konkrétne počas fázy preusoporiadávania častíc, pri prideľovaní lokálnych častíc každému procesu.

Implementovaná bola neblokujúca komunikácia v metódach *mpiAllGatherParticles* a *mpiAllGatherPosVel*. Iniciácia komunikácie bola umiestnená na mieste volania blokujúcich komunikačných funkcií. Synchronizáciu (*MPI_Waltall*) sme umiestnili bezprostredne na mieste, kde dané dáta boli nevyhnutné pre ďalšie výpočty. Avšak, zavedenie neblokujúcej komunikácie neprinieslo žiadne zrýchlenie oproti blokujúcej verzii. Veľkosť práce počas týchto prenosov je nedostatočná a tiež dochádzalo k prekrývaniu rôznych komunikácií. Preto v ďalšej kapitole boli vykonané testovania s využitím blokujúcej komunikácie.

Kapitola 8

Porovnanie výkonu behu N-Body simulácie v rámci klastra

Predchádzajúca kapitola sa zaoberala implementáciou, optimalizáciou a meraním výkonností zvolených algoritmov. Výkonnosť meraní bola prezentovaná len na úrovní jedného procesu. Zachytená bola zmena vo výkone spôsobená aplikovaním rôznych optimalizácií a zavedením paralelizmu na niekoľkých úrovniach. Zmeny vo výkonnosti algoritmov boli namerané len pri behu aplikácie v rámci jedného procesu. Taktiež sme porovnali výkonnosť algoritmu pri behu na procesore a koprocesore. Táto kapitola nadväzuje na merania z predchádzajúcej časti.

Ďalším krokom je testovanie výkonnosti zvolených algoritmov v distribuovanom prostredí klastra Salomon. Konfigurácia behu programov a metóda meraní je popísaná v nasledujúcej časti. Následne sú predstavené jednotlivé výsledky meraní s popisom. Jednotlivé merania sú zhodnotené a porovnané. Kapitola je zavŕšená celkovým zhodnotením nameraných výsledkov a tiež porovnaním jednotlivých konfigurácií behu programov v distribuovanom prostredí.

Metóda merania Cieľom tejto práce je porovnať výkonnosť algoritmov pri behu na zväzku procesorov, koprocesorov a v hybridnom móde, kedy sú využité procesory a aj koprocesory súčasne. Z toho vyplývajú tri hlavné testovacie konfigurácie:

- Beh na zväzku procesorov
- Beh na zväzku koprocesorov
- Beh na zväzku procesorov a procesorov - hybridný mód

Každá z týchto konfigurácií bola testovaná s niekoľkými rôznymi nastaveniami. Prvým nastavením bol počet uzlov výpočtové klastra, na ktorých boli testované nami implementované MPI aplikácie. Zvolili sme nasledovné hodnoty: 2 uzly (dva procesory, 4 a 8. Tabuľka 8.1 zobrazuje použité počty zariadení.

Ďalším parametrom bol počet procesov na zariadenie (procesor alebo koprocesor). Na základe meraní v predchádzajúcej kapitole boli zvolené hodnoty 1 a 2. Celkový počet vlákien na jednom uzle bol stále nastavený na počet jadier. Pri použití 2 procesov na zariadenie, v konfigurácií behu len na koprocesoroch a použití 8 výpočtových uzlov, dostávame celkový počet 32 procesov na 16 koprocesoroch. Počty výpočtových uzlov a procesov na zariadenie boli zvolené relatívne nízke z dôvodu využitia komunikácií typu každý s každým (napr.

Tabuľka 8.1: Počet procesorov, respektíve koprocesorov, dostupných pre beh MPI aplikácie pri rôznom počte výpočtových uzlov.

Počet uzlov klastra	Počet CPU	Počet MIC	Spolu
2	2	4	6
4	4	8	12
8	8	16	24

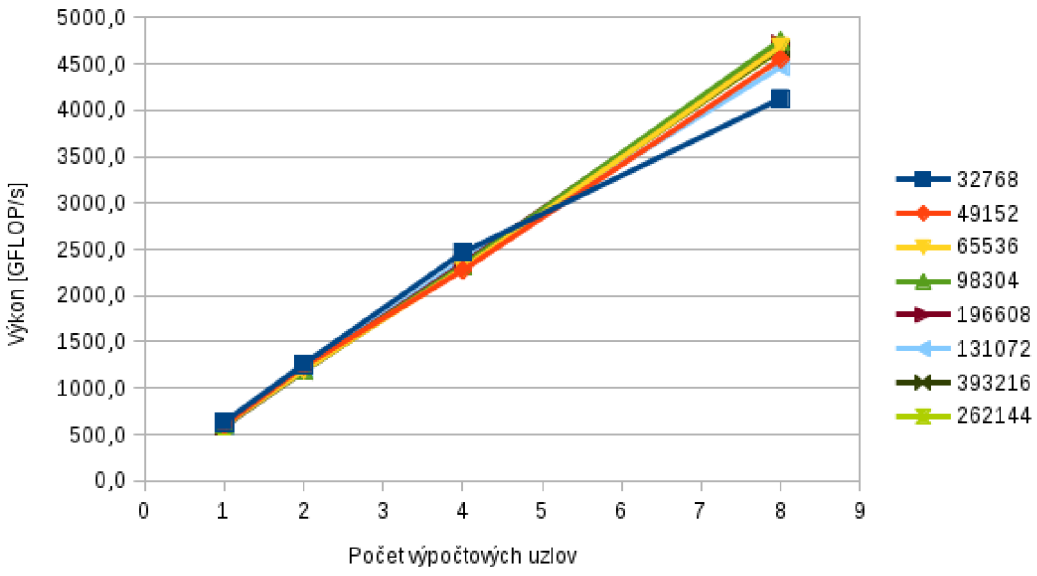
MPI_Allgather), pri ktorých s rastúcim počtom uzlov, respektíve procesov, rastie latencia, ako to vyplýva z meraní výkonnostných testov - OSU.

Čo sa týka testovacích vstupov, vygenerovaných bolo 8 datasetov s rôznym počtom častíc od $32K$ po $400K$. Hodnoty počtu častíc predstavujú násobky celkových počtov procesov v jednotlivých konfiguráciách s hodnotami 4096 a 8192. Pri počte častíc 4096 a 8192 boli dosiahnuté najvyššie namerané výkony pri behu programu v rámci jedného procesu.

Pri konfigurácií behu programu len na procesoroch bol odmeraný výkon programov aj pri behu na jednom uzle (pre každý dataset). Toto meranie bolo použité ako základ pre odhalenie škálovania programov v rámci klastra. Všetky namerané výsledky boli porovnané s výsledkami behu na jedno uzle s využitím jedného procesora. Toto porovnanie slúži na porovnanie a určenie zrýchlenia algoritmu pri zmene konfigurácie.

8.1 Naivná N-Body simulácia

Beh na procesoroch Prvým meraním bolo meranie výkonnosti naivnej verzie N-Body simulácie. Namerané hodnoty sú v grafe 8.1. Meranie pri počte výpočtových uzlov na úrovni 1 bolo použité ako referencia pri porovnávaní výkonu ostatných meraní.

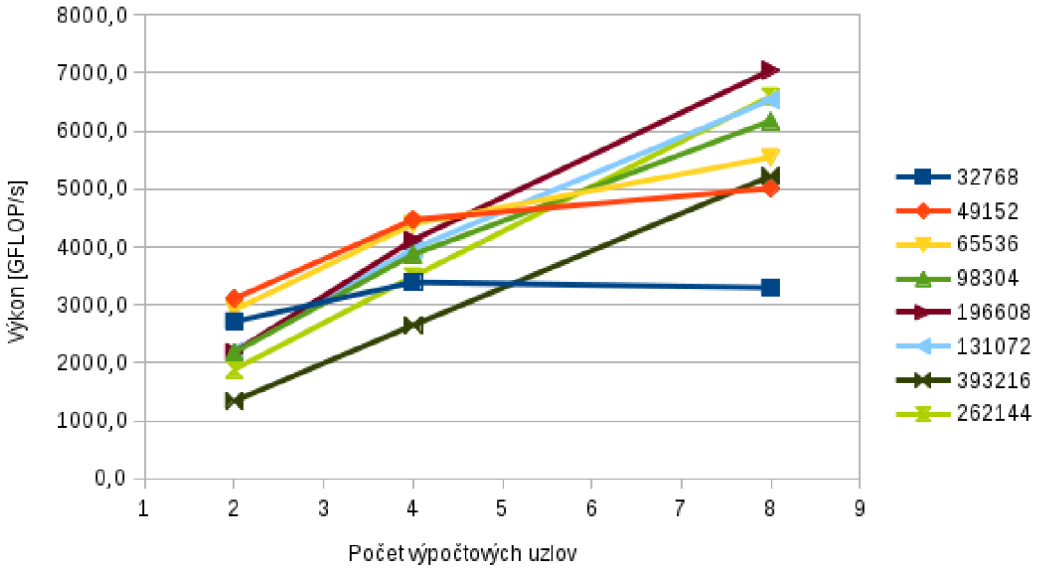


Obr. 8.1: Výkonnosť naivnej N-Body simulácie na procesore v rámci výpočtového klastra. Na každom uzle beží jeden proces.

Rovnako ako v prípade merania v časti 7.1.4 sme dosiahli úroveň 600 GFLOP/s pri behu

na jednom uzle. Maximálny výkon, 4, 7 TFLOP/s, bol dosiahnutý pri vstupe o veľkosti 98304 častíc pri behu programu na 8 uzloch. V porovnaní s behom algoritmu na jednom uzle o rovnakom vstupe došlo k zrýchleniu 7, 68. V ostatných prípadoch, okrem najmenšieho vstupu, výkon algoritmu sa pohyboval na úrovni 4, 5 TFLOP/s. V prípade 2 sa zrýchlenie blížilo k úrovni 2 a v prípade 4 bolo na úrovni 3, 9. Pri použití dvoch procesov na uzol boli dosiahnuté rovnaké výsledky.

Beh na koprocesoroch Graf na obrázku 8.2 zobrazuje namerané hodnoty výkonu pri behu programu na koprocesoroch. Zachytené sú merania pri jednom procese na koprocesor a teda dva procesy na uzol.



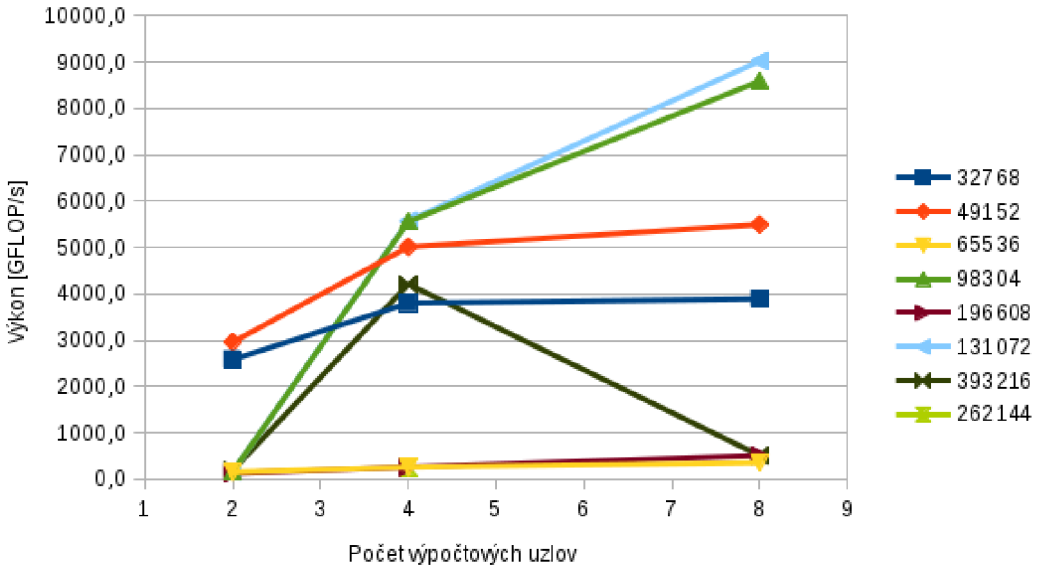
Obr. 8.2: Výkonnosť naivnej N-Body simulácie na koprocesore v rámci výpočtového klastra. Na každom koprocesore beží jeden proces (dva procesy na uzol).

Najvyšší výkon bol dosiahnutý pri behu na 8 uzloch (16 koprocesorov) a vstupe 196608 častíc - 7 TFLOP/s. S rastúcim počtom uzlov vo väčšine prípadov pozorujeme takmer lineárny nárast výkonu. Pri najmenších vstupoch sme pri použití 8 uzlov zaznamenali stagnáciu výkonu. Taktiež pri príliš veľkých vstupov bol celkový výkon o niečo nižší ako v prípade ostatných vstupov.

Pri použití konfigurácie s 2 procesmi na koprocesor sme zaznamenali pokles výkonu algoritmu. Graf s nameranými hodnotami pri dvoch procesoch na koprocesor je možné nájsť v prílohe C na obrázku C.2. Na základe meraní OSU výkonnostného testu predpokladáme, že zavedením dvojnásobného počtu procesov sa tiež zvýšia latencia kolektívnych MPI operácií, čo vo výsledku spôsobí pokles celkového výkonu algoritmu.

V porovnaní algoritmu pri behu na procesore, skupina koprocesorov na rovnakom počte uzlov dosahuje vyšší výkon. Maximálne dosiahnuté zrýchlenie bolo takmer 12 voči behu algoritmu na jednom uzle na procesore. V prípade behu len na procesoroch bolo zrýchlenie na úrovni blízko 8. Pri väčšine daných vstupov (okrem jedného), koprocesory dosiahli celkovo lepšie výsledky.

Hybridný mód Poslednou konfiguráciou je beh programu na procesoroch a koprocesoroch zároveň. Graf na obrázku 8.3 znázorňuje meranie výkonu v hybridnom móde. Ide o konfiguráciu s jedným procesom na výpočtovú jednotku, spolu 3 procesy na uzol.



Obr. 8.3: Výkonnosť naivnej N-Body simulácie na koprocesore v rámci výpočtového klastra. Na každom koprocesore beží jeden proces (dva procesy na uzol).

Najvyšší zaznamenaný výkon na úrovni 9 TFLOP/s sme zaznamenali pri maximálnom počte uzlov a vstupe 131072. V porovnaní s behom algoritmu na jednom uzle na procesore, došlo v tomto prípade k 15-násobnému zrýchleniu. Ak porovnáme najlepší prípad pri behu programu len na procesoroch a hybridný mód, výkon vzrástol takmer 2-násobne.

V grafe môžeme pozorovať neobvyklú degradáciu výkonu pri niektorých vstupoch a pri použití 2 uzlov. Meranie bolo niekoľko krát opakované s rovnakými výsledkami. Príčinu sa nám nepodarilo odhaliť. Pri použití 2 procesov na výpočtovú jednotku sa výkon v najhorších prípadoch z posledného merania mierne zlepšil, ale tiež maximálny dosiahnutý výkon klesol na úroveň $8,2$ TFLOP/s. Graf z tohto merania je možné nájsť v prílohe C na obrázku C.3.

8.1.1 Zhodnotenie

Naivná verzia N-Body simulácie bola implementovaná a optimalizovaná tak, že dokáže využiť potenciál SIMD jednotiek. Ide o problém, ktorý je možné jednoducho optimalizovať. Program bol najprv plne optimalizovaný, v rámci našich možností, pre procesor. Pri behu programu na jednom uzle sme namerali výkon na úrovni polovice maximálneho teoretického výkonu procesora ako je popísané v časti 3. Pre zvolené testovacie vstupy, výkon pri použití len procesorov sa s rastúcim počtom uzlov zvyšoval takmer lineárne.

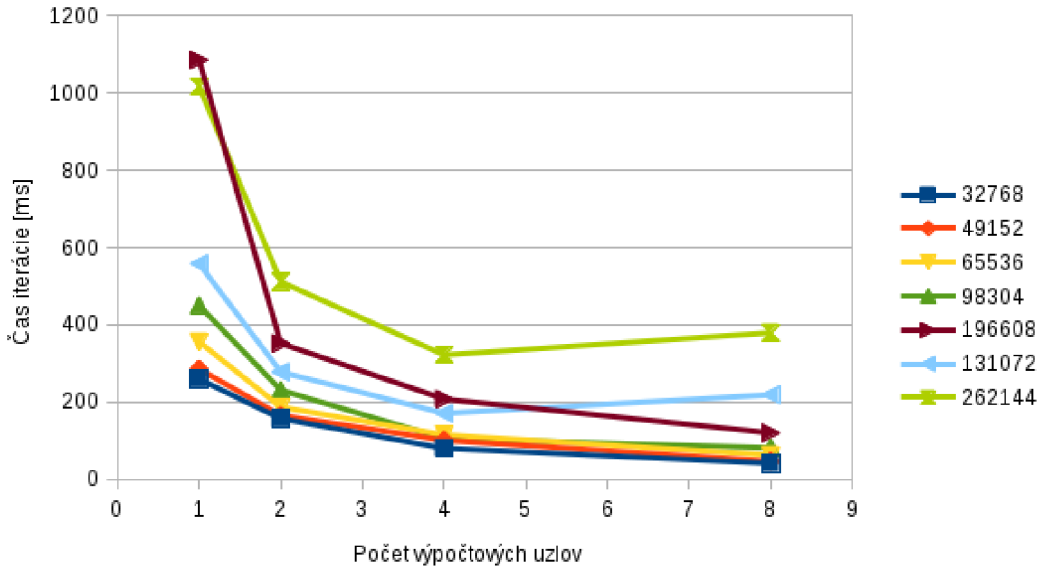
Programy, ktoré dokážu efektívne využiť SIMD jednotky sú vhodným kandidátom pre beh na koprocesore. Výkon na koprocesore je ale závislý na veľkosti vstupu. Pri behu programu na jednom koprocesore sme dosiahli v najlepšom prípade, podobne ako pri procesore, výkon na úrovni polovice maximálneho teoretického výkon. Pri prepočte počtu častíc na proces sa ukazuje, že koprocesor dosahuje maximálny výkon pri $8000 - 16000$ časticiach na proces. So zvyšujúcim sa počtom častíc tiež rastie množstvo komunikácie medzi procesmi.

Pri použití hybridného módu, bez akýchkoľvek zásahov do programu sa nám pri použití určitých vstupov podarilo získať zrýchlenie oproti predchádzajúcim konfiguráciám. Závislosť na veľkosti vstupných dát, respektíve na počte častíc na proces, sa v tomto prípade prejavila výraznejšie ako pri behu len na koprocesore. V porovnaní s behom len na koprocesore a len na procesore sa taktiež v hybridnom móde zvýšil počet procesov. Čo mohlo prispieť k ďalšiemu zvýšeniu latencie pri komunikácií medzi procesmi.

8.2 Barnes-Hut N-Body simulácia

Všetky inštancie merania výkonu Barnes-Hut N-Body simulácie boli spustené s rovnakými parametrami. Parameter θ bol nastavený na hodnotu $0,5$. Bola použitá blokujúca komunikácia. V týchto meraniach, ako ukazovateľ, výkonu bol zvolený čas jednej iterácie. Zrýchlenie Barnes-Hut algoritmu oproti naivnému prístupu spočíva vo vynechaní časti výpočtov interakcií. Pri rôznych vstupoch je počet vypočítaných interakcií častíc rôzny (viď tabuľka 7.5) a teda nebolo možné určiť konzistentne výkon medzi behmi programu s rôznym počtom vstupných častíc.

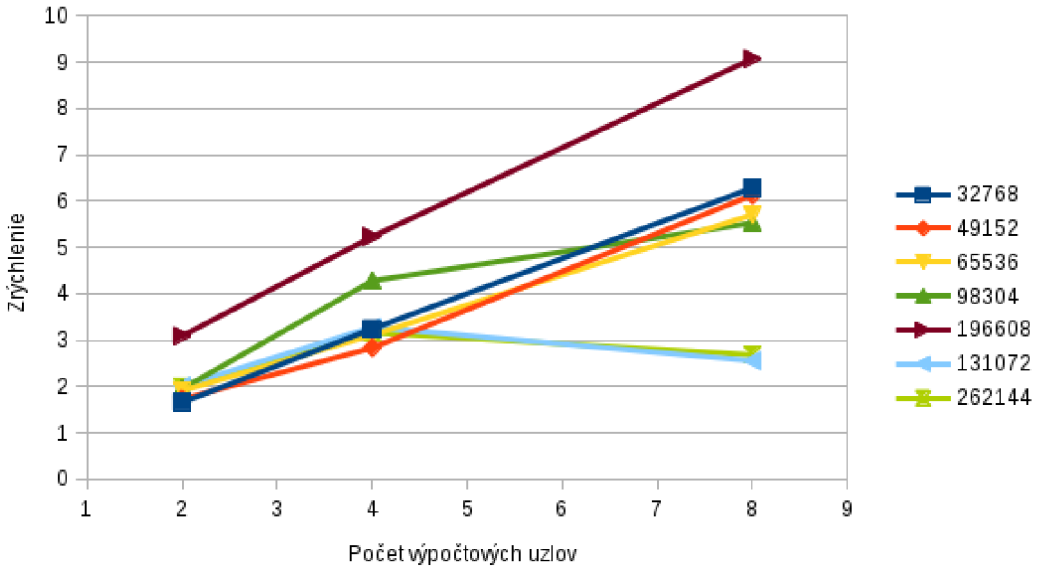
Beh na procesoroch Graf na obrázku 8.4 znázorňuje namerané časy behu jednej iterácie. Tento graf zachytáva beh programu pri 1 procese na procesor (uzol). Podobne ako v prípade naivnej verzie, namerané časy behu programu na jednom uzle boli použité ako referencia pri meraní dosiahnutého zrýchlenia použitím rôznych konfigurácií a tiež zvyšujúcim sa počtom výpočtových uzlov.



Obr. 8.4: Namerané časy behu jednej iterácie Barnes-Hut simulácie na procesoroch.

Na grafe môžeme pozorovať zníženie času iterácie s rastúcim počtom uzlov pri nižších datasetoch. V niektorých prípadoch (počet častíc 131072 a 262144) došlo k zníženiu výkonu pri zavedení 4 ďalších uzlov oproti behu na 4 uzloch. Pri vstupe 196608 nepozorujeme takýto pokles výkonu. Predpokladáme, že hlavnou príčinou degradácie výkonu pri použití 8 uzlov

je nerovnomerné rozloženie práce medzi jednotlivé procesy a do istej miery tiež zvýšenie latencie zavedené zvýšeným počtom procesov.



Obr. 8.5: Namerané zrýchlenie behu Barnes-Hut simulácie na procesoroch.

Škálovanie výkonu programov je zobrazené v grafe 8.5. Zrýchlenie je určené ako pomer času behu programu na jedno v uzle a času behu programu na príslušnom počte uzlov s rovnakými parametrami. Pri veľkých vstupoch zvýšenie počtu výpočtových uzlov neprinieslo očakávané zrýchlenie. Vo väčšine prípadov bolo dosiahnuté zrýchlenie na úrovni 5 až 6 oproti behu na jednom uzle.

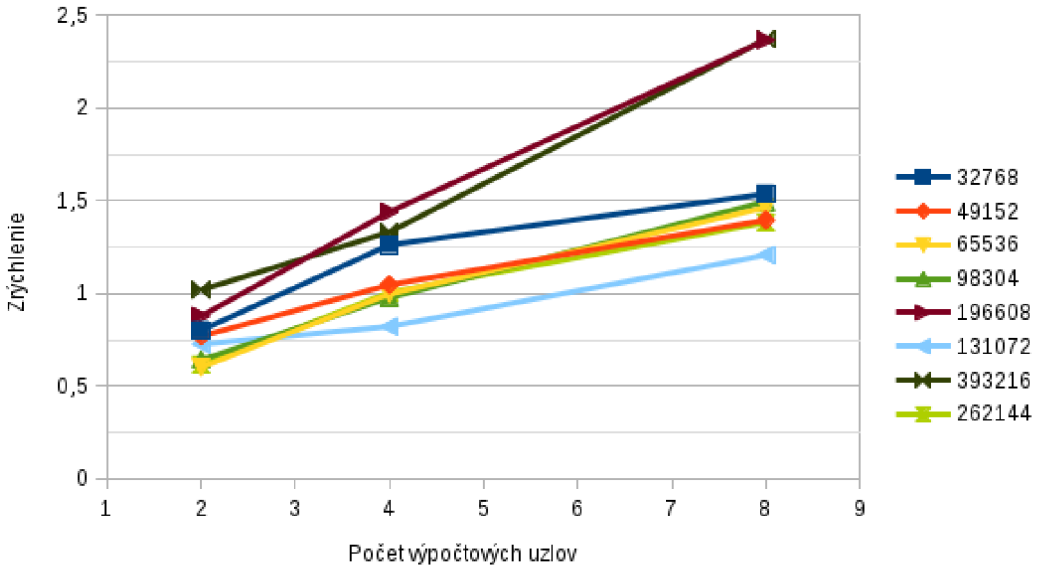
Beh na koprocesoroch Obrázok 8.6 obsahuje graf zachytávajúci namerané zrýchlenie programu pri behu na koprocesore. Zrýchlenie je počítané vzhľadom na namerané časy behu programu na procesore na jednom výpočtovom uzle. Ide o konfiguráciu s jedným procesom na koprocesor. Namerané časy jednotlivých meraní na koprocesoroch je možné nájsť na obrázku D.2 v prílohe D.

Čas jednej iterácie sa podľa očakávaní zvyšuje s rastúcou veľkosťou vstupu. Podobne ako v prípade behu na procesoroch, aj tu pozorujeme pokles výkonu programu pri vstupe 131072.

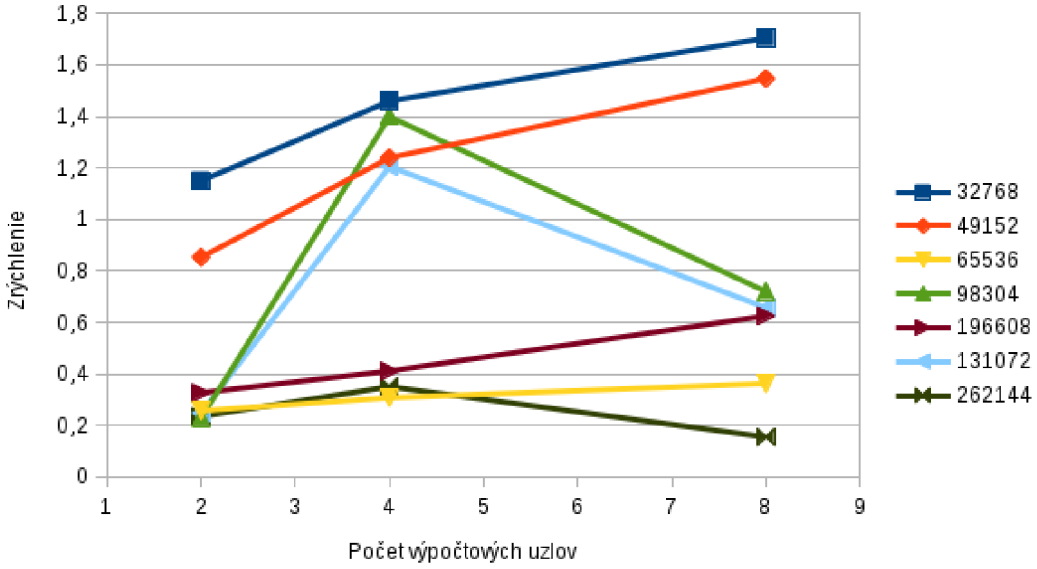
Hybridný mód Graf 8.7 zachytáva namerané zrýchlenie pri behu v hybridnom móde pri použití jedného procesu na výpočtovú jednotku (procesor/koprocesor). V konfigurácií 4 výpočtových uzlov došlo k neobvyklému zníženiu času jednej iterácie pri niektorých vstupoch.

8.2.1 Zhodnotenie

Implementovaný Barnes-Hut algoritmus nevyužíva efektívne SIMD jednotky, čo sa odzrkadlilo na dosiahnutých výkonoch programu. Značnú časť výpočtu zaberá prechod stromom a výpočet síl. Z princípu fungovania algoritmu vyplýva, že pri prechode stromom pre každú



Obr. 8.6: Namerané časy behu jednej iteráci Barnes-Hut simulácie na koprocesoroch.



Obr. 8.7: Namerané časy behu jednej iterácie Barnes-Hut simulácie v hybridnom móde.

časticu môže dôjsť k predčasnému ukončeniu prehľadávania podstromu uzlu, čo ma za následok nerovnomerné rozdelenie práce pri pravidelnom rozdelení vstupného datasetu. Táto nevyváženosť sa prejavuje na viacerých úrovniach. Prvou úrovňou je rozdelenie práce medzi MPI procesy. Každý proces má pridelenú časť častíc pre ktorú počíta sily. Druhou úrovňou je rozdelenie práce medzi vlákna v rámci procesu.

Prirodzeným predpokladom je, že s rastúcou veľkosťou vstupu rastie tiež aj čas na spracovanie tohto vstupu. Pri behu Barnes-Hut algoritmu na procesoroch (graf 8.4) pozorujeme, že pri väčšom vstupe bol čas menší ako pri o niečo menšom vstupe. V tomto prípade nameraný čas bol ovplyvnený počtom interakcií, ktoré boli vynechané použitím oktálového stromu.

Porovnávanie výkonnosti algoritmu na základe doby behu jednej iterácie nie je príliš možné. Z grafov je možné odčítať absolútne časy behu programov. Ako ukazovateľ zmeny výkonu pri zväčšení počtu uzlov, respektíve behu na a s využitím koprocesorov bol použitý pomer časov jednej iterácie pri behu na jednom uzle na procesoroch k časom jednej iterácie pri danom vstupe.

Behom programu na zväzku koprocesorov a tiež v hybridnom móde nebolo dosiahnuté požadované zrýchlenie. Hlavným nedostatkom tohto algoritmu bolo slabé využitie SIMD jednotiek. Ďalším faktorom, ktorý ovplyvnil výsledné zrýchlenia je tiež nedostatočné rozloženie záťaže medzi procesmi.

8.3 Porovnanie

Implementované a optimalizované verzie N-Body simulácie sme porovnali vzhľadom k dosiahnutému zrýchleniu oproti behu programov na jednom uzle. Účelom tohto porovnania je charakterizácia možností klastra. Naivná verzia reprezentuje problém, respektíve program, v ktorom je možné plne využiť dostupná zdroje výpočtovej jednotky (všetky jadrá, SIMD jednotky). Na druhej strane, nami implementovaná verzia Barnes-Hut N-Body simulácie predstavuje problém so slabým využitím dostupných zdrojov výpočtovej jednotky.

V oboch prípadoch pri behu na procesoroch pozorujeme zvýšenie výkonu s rastúcim počtom uzlov. Barnes-Hut verzia však trpí poklesom výkonu v niektorých prípadoch. Vyššie celkové zrýchlenie bolo dosiahnuté pri dobre optimalizovanom programe s plným využitím SIMD jednotiek. V prípade Barnes-Hut simulácie bolo dosiahnuté zrýchlenie aj pri neoptimálnom využití dostupných zdrojov. Graf nameraných zrýchlení je možné nájsť v prílohe C na obrázku C.1. V prvom prípade bolo dosiahnuté zrýchlenie na úrovni 6 až 7. U Barnes-Hut verzie sa zrýchlenie pohybovalo na úrovni 5 až 6 s maximom na úrovni 9. Toto pomerne vysoké zrýchlenie je zapríčinené veľmi nízkym výkonom pri behu na jednom procesore s daným vstupom a následným prudkým nárastom výkonu.

Presunutím programov na koprocesory, rozdiely medzi zrýchlením na procesoroch sa začali viac prehlbovať. Pri Barnes-Hut verzii sme zaznamenali prudký pokles výkonu a tiež absolútnych časov jednej iterácie. Táto degradácie výkonu demonštruje nevhodnosť behu programov s nízkym využitím SIMD paralelizmu. Na druhú stranu, naivná verzia N-Body vykazuje ďalší nárast výkonu v porovnaní s behom na jednom uzle na procesore. Išlo o zrýchlenie 8 až 12 v niektorých prípadoch. Barnes-Hut N-Body simulácie dosiahla zrýchlenie len 1 až $2,5$ pri použití 8 uzlov.

Využitím procesorov a koprocesorov s hybridnom móde sme pozorovali nestabilitu vo výkone oboch implementácií. Výkon do veľkej miery bol závislý na vstupných dátach. Prehlbovanie rozdielu pozorovaného pri behu len na koprocesoroch pokračovalo aj v tejto konfigurácií. V najlepších prípadoch, dosiahnuté zrýchlenie Barnes-Hut verzie kleslo na ešte

nižšiu úroveň a naivná verzia dosiahla o niečo lepšie zrýchlenie ako v predchádzajúcom behu.

8.4 Zhodnotenie výhod jednotlivých architektúr

Na základe nameraných údajov v tejto práci sme dospeli k nasledujúcemu zhodnoteniu výhod a nevýhod zväzku procesorov a koprocesorov pri použití s MPI. Vychádzali sme z výkonnostných testov popísaných v kapitole 6 a z nami implementovaných N-Body simulácií. Výhody a nevýhody sú prezentované vo vzťahu k ostatným konfiguráciám.

8.4.1 Zväzok procesorov

Výhody

- Dosiahnuteľné zrýchlenie aj pri neefektívnom využití SIMD jednotiek
- Vyššia dostupná šírka pásma pri použití MPI komunikácie
- Nižšia latencia pri použití MPI komunikácie

8.4.2 Zväzok koprocesorov

Výhody

- Vyšší výkon pri efektívnom využití výpočtových zdrojov
- Vyššia dostupná celková šírka pásma pamäte
- Jednoduché portovanie optimalizovaných aplikácií pre Intel Xeon CPU.

Nevýhody

- Veľmi malé dosiahnuteľné zrýchlenie pri neefektívnom využití SIMD.
- Menšie variácie vo výkone pri zmene veľkosti vstupných dát.
- Nie príliš vhodné pre beh program nevyužívajúcich SIMD.

8.4.3 Hybridný mód

Výhody

- Dosiahnuteľné zrýchlenie pri dobre optimalizovaných aplikáciach.

Nevýhody

- Variácie vo výkone pri zmene veľkosti vstupu.
- Nevhodné pre beh program nevyužívajúcich SIMD.
- Potreba implementácie rozloženia práce medzi CPU a MIC.

Kapitola 9

Záver

Cieľom tejto práce bolo porovnať výkonnosť implementácií zvolených problémov bežiacich na zväzku procesorov, zväzku koprocesorov a v hybridnom móde s využitím oboch typov hardvéru. Ako demonštračné problémy sme zvolili dve verzie N-Body simulácie. Naivnú verziu a Barnes-Hut variáciu N-Body simulácie, ktorá aproximuje výpočet síl pôsobiacich na častice. Obe verzie N-Body simulácie boli implementované a následne optimalizované.

Naivná verzia N-Body simulácie predstavuje problém, ktorý je dobre paralelizovateľný. Implementovaný program efektívne využíva SIMD jednotky a tiež paralelizmus na úrovni vlákien. V najlepšom prípade bol dosiahnutý maximálny výkon na procesore 565 GFLOP/s. Presunutím výpočtu na koprocesor sme namerali maximálny výkon na úrovni 1 TFLOP/s. Dosiahnutý výkon bol závislý na veľkosti vstupu. Na základe nameraných dát sme určili, že v tomto prípade, počet častíc, pre dosiahnutie optimálneho výkonu sa pohybuje na úrovni $8000 - 14000$ častíc.

Na druhú stranu, Barnes-hut verzia N-Body simulácie predstavuje problém, ktorý nie je triviálny na optimalizáciu. V tomto prípade sa nám nepodarilo zabezpečiť nárast výkonu presunutím výpočtu na koprocesor.

V ďalšej fáze vývoja práce boli programy upravené pre beh v distribuovanom prostredí. Testovanie a merania prebiehali na niekoľkých konfiguráciách (beh na procesoroch, koprocesoroch a na oboch zároveň). Podľa očakávaní, naivná verzia N-Body simulácie vykazovala nárast výkonu vo všetkých prípadoch pri zvyšujúcom sa počte výpočtových uzlov. Najvyšší nameraný výkon sme zaznamenali pri použití 8 výpočtových uzlov pri behu programu v hybridnom móde. Výkon sa pohyboval na úrovni 9 TFLOP/s. Avšak pri behu v hybridnom móde sme taktiež pozorovali nestabilitu výkonu. Pri niektorých vstupoch dochádzalo k degradácií výkonu. Pri meraniach Barnes-Hut verzie sme zaznamenali nárast výkonu oproti behu na jednom uzle len v prípade behu na procesoroch. Pri zapojení koprocesorov do výpočtu, výkon začal prudko klesať.

Na základe nameraných dát sme určili výhody a nevýhody jednotlivých architektúr. Problémy, v ktorých je možné efektívne využiť SIMD jednotky v kombinácií s paralelizmom na úrovní vlákne sú vhodné pre beh na koprocesoroch, respektíve v hybridnom móde. Predpokladáme, že zavedením lepšieho rozdelenia práce medzi procesy a minimalizáciou komunikácie je možné dosiahnuť ešte lepšie výsledky. Pri problémoch, ktoré nevyužívajú SIMD jednotky, behom na koprocesoroch veľa nezískame. Môže dôjsť až k zníženiu výkonu oproti zväzku procesorom.

Literatúra

[1] Besl, P.: A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel®Xeon®processors and Intel®Xeon Phi™product family coprocessors [online]. 2013 [cit. 2016-05-24].
URL https://software.intel.com/sites/default/files/article/392271/aos-to-soa-optimizations-using-iterative-closest-point-mini-app.pdf

[2] Blelloch, G.; Narlikar, G.: *A Practical Comparison of N-Body Algorithms*. American Mathematical Society, 1997.

[3] Bédorf, J.; Gaburov, E.; Fujii, M. S.; aj.: 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs. *CoRR*, ročník abs/1412.0659, 2014.

[4] Bédorf, J.; Gaburov, E.; Zwart, S. P.: A sparse octree gravitational N-body code that runs entirely on the GPU processor. *CoRR*, ročník abs/1106.1900, 2011 [cit. 2016-05-24].

[5] Corden, M.: Requirements for Vectorizing Loops with #pragma SIMD [online]. 2015 [cit. 2016-05-24].
URL https://software.intel.com/en-us/articles/requirements-for-vectorizing-loops-with-pragma-simd

[6] Intel: Interoperability with OpenMP API [online].
https://software.intel.com/en-us/node/528819, [cit. 2016-05-24].

[7] IT4I: Intel Xeon Phi - A guide to Intel Xeon Phi usage [online].
https://docs.it4i.cz/salomon/software/intel-xeon-phi, [cit. 2016-05-24].

[8] It4Inovations: Salomon Cluster Documentation - Hardware Overview [online]. [cit. 2016-05-24].
URL https://docs.it4i.cz/salomon/hardware-overview-1

[9] Jeffers, J.; Reinders, J.: *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013, ISBN 978-0-12-410414-3.

[10] Nguyen, H.: *Gpu Gems 3*. Addison-Wesley Professional, první vydání, 2007, ISBN 9780321545428.

[11] OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0 [online]. 2013 [cit. 2016-05-24].
URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Prílohy

Zoznam príloh

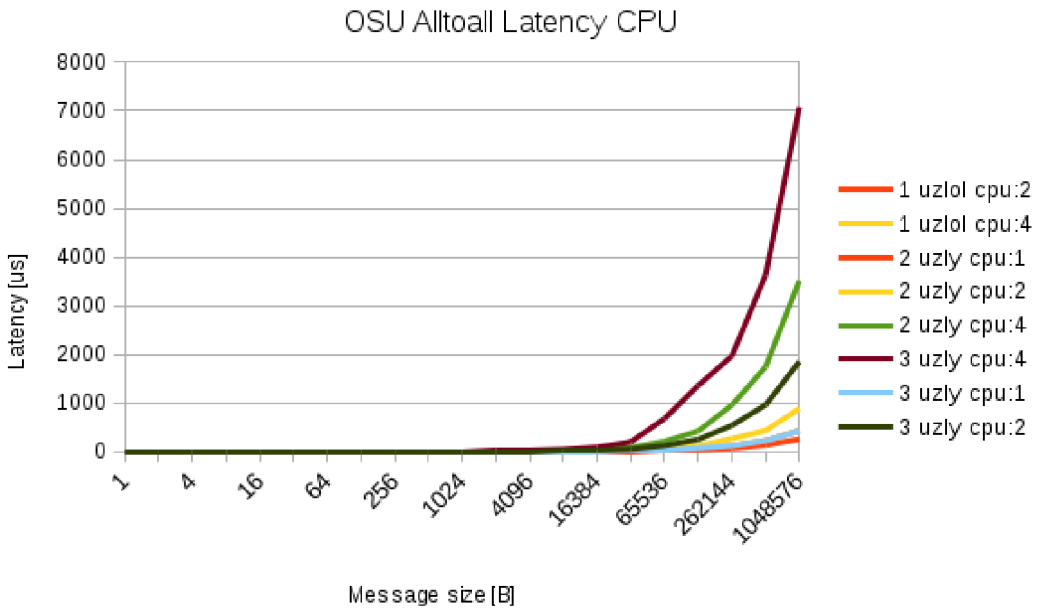
Príloha A

Obsah CD

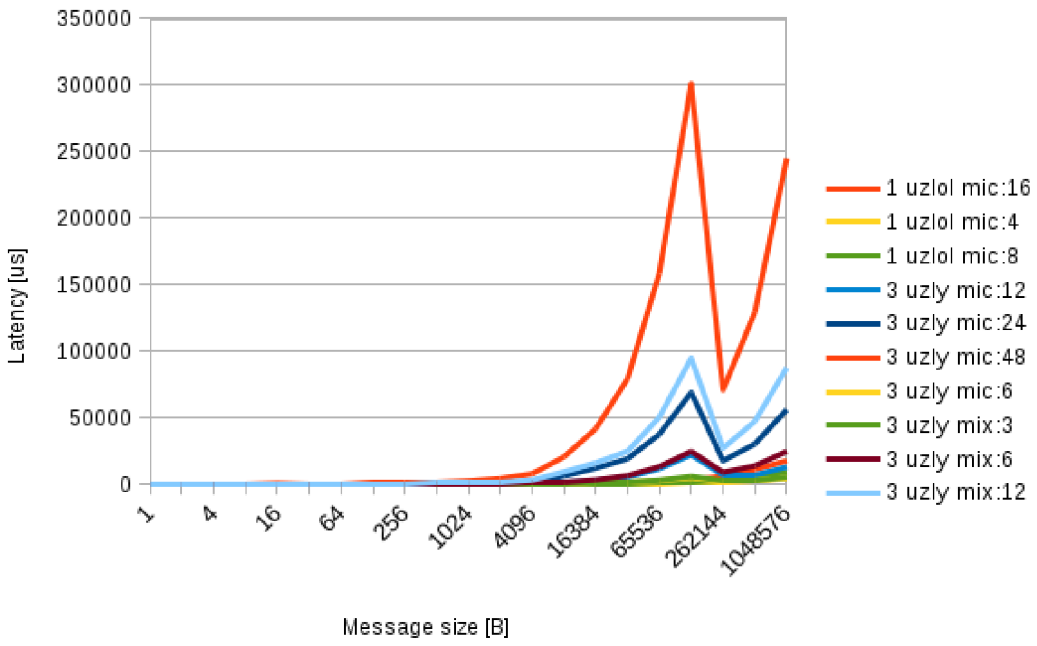
- **matvec/** - zdrojové kódy matvec výkonnostného testu a merania.
- **matvec-mkl/** - zdrojové kódy matvec-mkl výkonnostného testu a merania.
- **nbody/** - zdrojové kódy N-Body implementácií a merania.
- **osu-micro-benchmark/** - skripty použité pri tomto výkonnostnom teste a merania.
- **stream/** - zdrojový kód výkonnostného testu a merania.
- **document/** - zdrojové kódy tohto dokumentu.
- **README** - popis obsahu na CD. Každá zložka ďalej obsahuje README - stručný popis obsahu danej zložky.

Príloha B

Grafy OSU výkonnostného testu



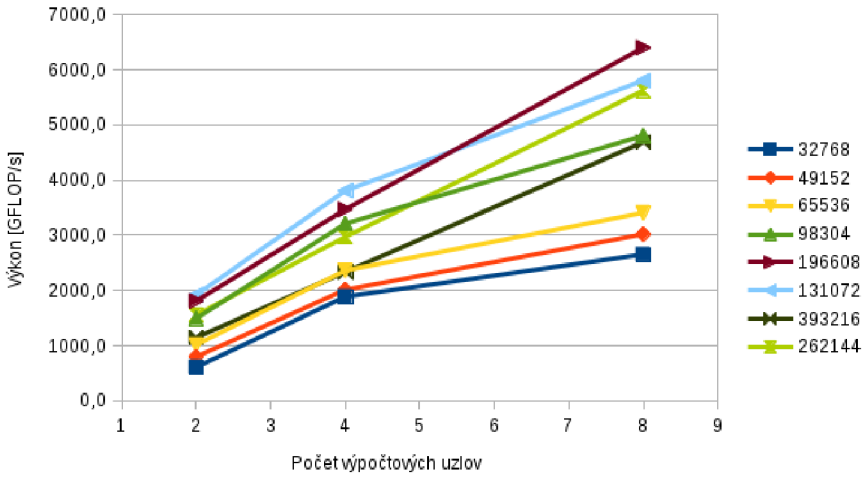
Obr. B.1: Namerané hodnoty výkonnostného testu OSU Alltoall - Latency na procesore.



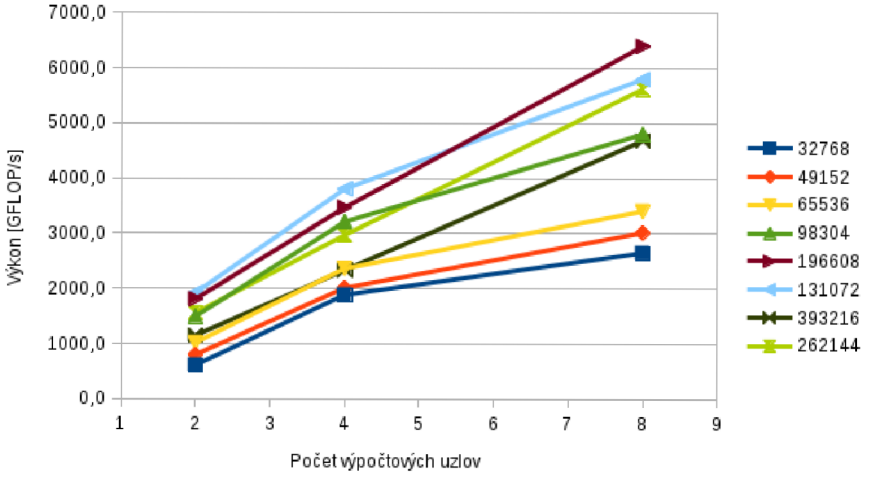
Obr. B.2: Namerané hodnoty výkonnostného testu OSU Alltoall - Latency na koprocesore.

Príloha C

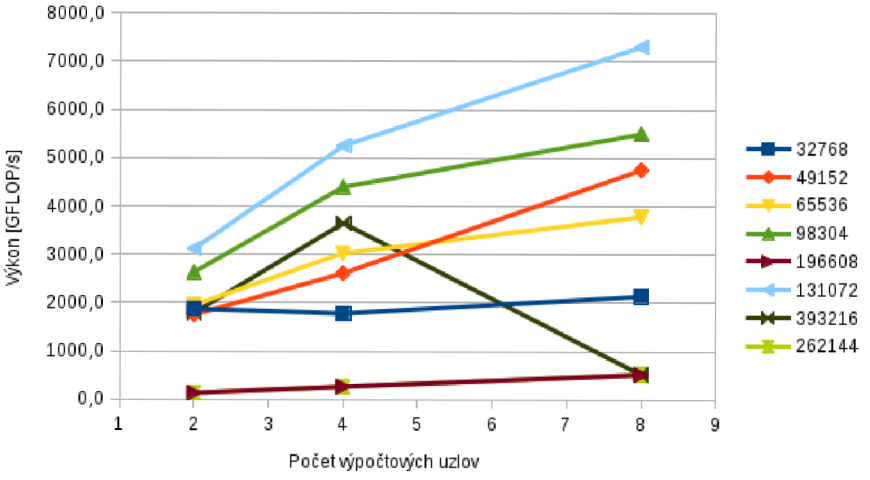
Grafy - naivná verzia



Obr. C.1: Graf dosiahnutého zrýchlenia pri behu na procesoroch.



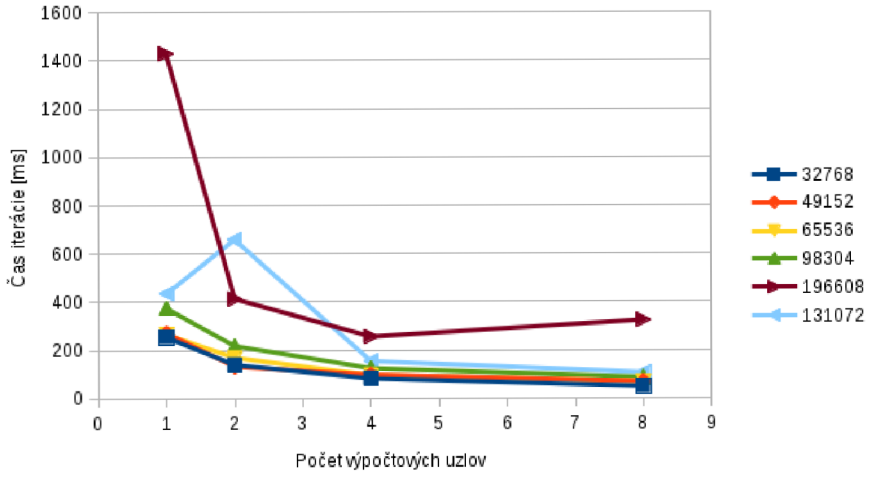
Obr. C.2: Výkonnosť naivnej N-Body simulácie na koprocesore v rámci výpočtového klastra. Na každom koprocesore bežia dva procesy (štyri procesy na uzol).



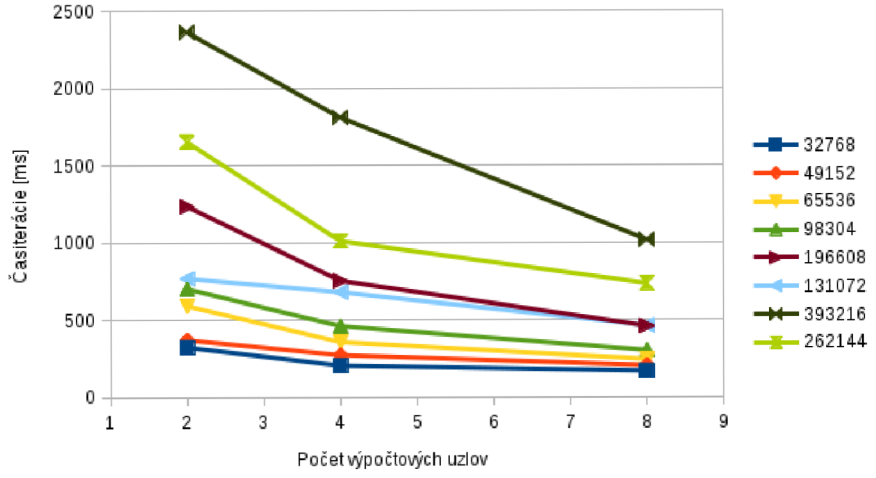
Obr. C.3: Výkonnosť naivnej N-Body simulácie pri behu v hybridnom móde. Na každom zariadení (procesorov/koprocesor) bežia dva procesy (šesť procesy na uzol).

Príloha D

Grafy - Barnes-But



Obr. D.1: Namerané časy pri behu na procesoroch s 2 procesmi na procesor.



Obr. D.2: Namerané časy behu jednej iterácie Barnes-Hut simulácie na koprocesoroch.