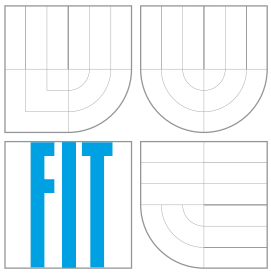
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EFEKTIVNÍ IMPLEMENTACE VYSOCE NÁROČNÝCH ALGORITMŮ NA VÍCEJÁDROVÝCH PROCESORECH

EFFICIENT IMPLEMENTATION OF HIGH PERFORMANCE ALGORITHMS ON MULTI-CORE PROCESSORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ TOMEČKO
AUTHOR

VEDOUCÍ PRÁCE — JAROŠ JIŘÍ, Ing., Ph.D.
SUPERVISOR

BRNO 2016

Abstrakt

Cieľom tejto práce je paralelizovať a vektorizovať simuláciu toku kvapalín. Dosiahne sa to pomocou knižnice OpenMP a prekladaču od Intelu. Implementované boli rôzne prístupy k problému, ako napr. cache blocking, zoraďovanie dát počas behu a dočasné reorganizovanie dát v pamäti.

Skombinovaním najrýchlejších riešení sa podarilo simuláciu celkovo zrýchliť 11,4krát na 16 jadrách, pričom testy prebiehali na ostravskom superpočítači Anselm. Výsledky ukazujú, že výsledná aplikácia dobre škáluje s pribúdajúcim počtom jadier. Ďalej, vektorizovanie daného problému bolo možné len čiastočne z dôvodu nevhodného spôsobu práce s dátami.

Abstract

This thesis describes the process of parallelization and vectorization of fluid simulation using OpenMP library and Intel compiler. Various approaches were tried e.g. cache blocking, data sorting and data reorganization. By combining the best of them, final application preformed 11.4 times faster than the original one, using 16 cores. Benchmarks show that used algorithms are not suitable for vectorization.

Klíčová slova

paralelizácia, vektorizácia, OpenMP, simd, Intel, SPH, simulácia, násobenie matíc

Keywords

parallelization, vectorization, OpenMP, simd, Intel, SPH, simulation, matmul

Citace

Lukáš Tomečko: Efektivní implementace vysoce náročných algoritmů na vícejádrových procesorech, bakalářská práce, Brno, FIT VUT v Brně, 2016

Efektivní implementace vysoce náročných algoritmů na vícejádrových procesorech

Prohlášení

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

.....
Lukáš Tomečko
17. mája 2016

Poděkování

V prvom rade by som chcel poďakovať môjmu vedúcemu práce doktorovi Jiřímu Jarošovi za jeho entuziazmus a trpezlivosť so mnou. Ďalej ďakujem Bc. Dominikovi Šimekovi za odborné diskusie.
Táto práca bola podporovaná Ministerstvom školstva, mládeže a telovýchovy z projektu Veľkých infraštruktúr pre výzkum, experimentálny vývoj a inovácie „IT4Innovations národní superpočítačové centrum - LM2015070“.

Obsah

Kapitola 1

Úvod

Ľudia sa už od prvých počítačov snažili zvyšovať ich výkon, čo sa dosahovalo hlavne zvyšovaním pracovnej frekvencie ich procesorov. Na prelome tisícročí však začali narážať na technologické limity. So zvyšujúcou frekvenciou sa zvyšovala aj spotreba procesorov a zvyškové teplo, ktoré bolo treba nejako odvádzať. Riešenie prinieslo zvýšenie počtu procesorov s nižšou frekvenciou, čo bolo možné aj vďaka neustále zmenšujúcemu sa výrobnému procesu. Viac výpočtových jednotiek znamenalo, že môže bežať viacero úloh súčasne, čo znamená teoreticky väčší výkon. [19]

Rýchlosť behu aplikácií však nezávisí len na hardvéri. Nevhodne napísaný sériový program nemusí dosahovať ani len percento potenciálneho výkonu daného počítača. Naproti tomu, jeho optimalizovaná paralelná verzia môže dosahovať aj zrýchlenie v rádoch desiatok či dokonca stoviek.

Táto práca sa preto zaoberá možnosťami modifikácie softvéru tak, aby využil schopnosti daného počítača na maximum. Dosiahne sa to pomocou paralelizácie a vektorizácie s ohľadom na cieľový počítač.

Z tohto dôvodu sa v kapitole 2 popisuje architektúra moderných procesorov, vrátane testovacej zostavy. V kapitolách 3 a 4 bude vysvetlená paralelizácia a vektorizácia s využitím knižnice OpenMP a Intel prekladača. Vysvetlená teória sa následne aplikuje v mikrotestoch (kapitola 5), ktorých cieľom je vyhodnotenie chovania testovacej zostavy. Nadobudnuté poznatky sa potom využijú na optimalizáciu praktickej úlohy v kapitole 6. Konkrétne, bude sa jednať o 2D simuláciu toku kvapalín.

Kapitola 2

Architektúra procesorov

Táto kapitola stručne popisuje konštrukciu a chovanie súčasných procesorov, čo je pre programátora efektívneho softvéru kľúčové. S požiadavkami na vyšší výkon procesorov sa zvyšovala aj ich komplexnosť. Preto sú v jednotlivých podkapitolách uvedené situácie, ktoré môžu negatívne vplývať na výkon. Na konci kapitoly je charakteristika stroja Anselm, na ktorom prebiehali testy. Táto kapitola vychádza z [15].

2.1 Prúdové spracovanie inštrukcií

Inštrukcie, tvoriace softvér, sa v súčasných procesoroch vykonávajú prúdovo (pipelining). To znamená, že nová inštrukcia sa začne vykonávať bez toho, aby sa čakalo na dokončenie tej predchádzajúcej. Vo výsledku sa tým zvýši počet vykonaných inštrukcií ale len za predpokladu, že procesor vie, ktorá inštrukcia nasleduje a má potrebné operandy.

V prípade, že operácia požaduje výsledok inštrukcie, ktorá sa ešte nedokončila, aktuálna operácia musí naň čakať, čím sa pozastaví prúd následných inštrukcií. Vyhnúť sa tomu dá zmenou poradia inštrukcií tak, aby výsledok zostal rovnaký. O to sa môže postarať prekladač pri optimalizáciách.

Ďalší prípad, kedy procesor musí čakať, je načítavanie dát z pamäte. Toto spomalenie sa redukuje využitím rýchlych vyrovnávacích pamätí (viac v kapitole 2.3).

Procesory, podporujúce viac hardvérových vlákien, dokážu v prípade takéhoto čakania pokračovať vo vykonávaní iného vlákna (napr. technológia Hyper-Threading od Intelu). Výkon čakajúceho vlákna sa síce nezlepší, ale zlepší sa priepustnosť procesoru. Pre vedecké výpočty sa odporúča takúto technológiu vypnúť. [14]

Aby sa mohli inštrukcie vykonávať bez pozastavenia, musí procesor vedieť, ktorá inštrukcia nasleduje. V prípade podmienených skokov nie je známe, či sa skok vykoná, kým sa nevyhodnotí podmienka, na ktorej vetvenie kódu závisí. Tento problém sa rieši predikciou, či sa skok vykoná. Podľa toho procesor pokračuje inštrukciami z predikovanej adresy. Ak bola predikcia nesprávna (branch misprediction), rozpracované inštrukcie sa zahodia a začne sa vykonávať správny sled inštrukcií. Kvôli tomuto spomaľovaniu je vhodné minimalizovať množstvo podmienených skokov.

2.2 Vektorové operácie

V súčasných procesoroch existujú tzv. SIMD inštrukcie (Single Instruction Multiple Data), s ktorými je možné vykonať jednu operáciu nad viacerými prvkami vektoru súčasne. V pro-

cesoroch sú tieto operácie prítomné ako rozšírenia inštrukčnej sady, napr. pre x86 existujú SSE a AVX, v prípade ARM to sú NEON. Počet a typy prvkov, ktoré sú schopné naraz spracovať, je závislý na konkrétnej implementácii, pričom dĺžky vektorov (resp. pracovných registrov) sa zvyčajne pohybujú medzi 128 až 512 bitov. [8]

Vektorizácia sa dá použiť aj v prípadoch, kedy sú spracovávané dáta príliš malé na efektívnu paralelizáciu. Tabuľka 2.1 ukazuje, aké je teoretické zrýchlenie vektorizovaného kódu oproti skalárnemu. V praxi sa však také zrýchlenie dosiahne len zriedka, viac v kapitole 4.

	Instruction Set	SP FLOPs per cycle	DP FLOPs per cycle
Nehalem	SSE (128-bits)	8	4
Sandy Bridge	AVX (256-bits)	16	8
Haswell	AVX2 & FMA	32	16

Obr. 2.1: Počet operácií s desatinnými číslami vykonateľnými za cyklus pre Intel mikroarchitekúry. FMA znamená fused multiply-add – násobenie a sčítanie v jednej inštrukcii. [5]

2.3 Cache

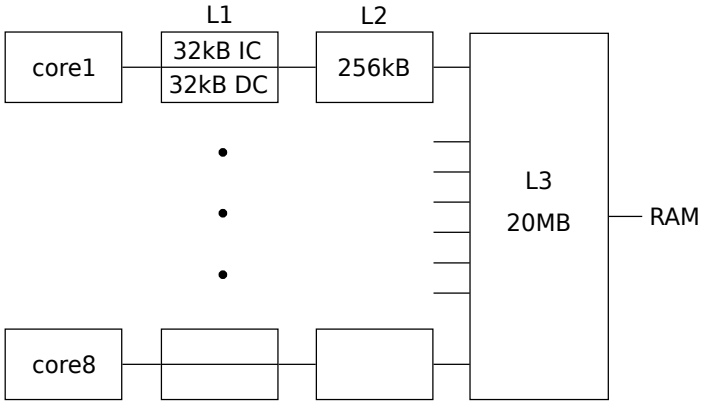
Cache, tiež nazývaná rýchla vyrovnávacia pamäť alebo medzipamäť, v kontexte procesoru vyrovnáva rýchlosti medzi rýchlym procesorom a pomalšou pamäťou (doba prístupu v stovkách cyklov procesoru). V procesoroch môžu byť až 3 úrovne týchto pamätí, nazývané L1, L2 a L3. Cache L1 je najbližšie ku procesoru, je najmenšia (niekoľko kB), ale zároveň najrýchlejšia, s dobou prístupu v jednotkách cyklov. Pamäť L3 je bližšie ku pamäti, jej veľkosť sa pohybuje v jednotkách MB ale je najpomalšia (latencia v desiatkach cyklov). Tieto pamäte sa ešte môžu rozdeliť aj v rámci jednej úrovne na inštrukčnú a dátovú cache.

Pri čítaní dát z pamäte sa dáta čítajú po tzv. riadkoch medzipamäti (cache line), ktorých veľkosť je typicky 64 bajtov. Pri načítaní riadku z pamäte alebo vzdialenejšej medzipamäti sa kópia tohto riadku zároveň ukladá do medzipamätí nižšej úrovne. Tým sa zvýši výkon, pretože je pravdepodobné, že rovnaké alebo dáta z rovnakého riadku sa znovu použijú, a teda ich už netreba čítať z pomalej hlavnej pamäte.

Ak sa požadované dáta nenachádzajú v L1 cache, dochádza k tzv. výpadku (cache miss) a dáta sa hľadajú vo vzdialenejších cache alebo dokonca až v pomalej hlavnej pamäti. [19] Preto je dôležité pri návrhu efektívneho softvéru myslieť na maximálne využitie medzipamätí.

2.4 Paralelné systémy

Predchádzajúce kapitoly popisovali chovanie procesoru, ktorý v každom okamžiku vykonáva práve jedno vlákno. Keďže v praxi beží v systéme viac vlákien súčasne, zvyšovanie celkového výkonu sa dosahuje zvyšovaním počtu jadier v rámci procesoru. Takéto jadrá počítajú nezávisle na sebe, pričom podľa architektúry zdieľajú spolu niektoré zdroje, napr. prístup do pamäte či L3 cache.



Obr. 2.2: Hierarchia pamätí v rámci jedného procesoru Anselmu.

Pre väčší výkon je možné zvýšiť počet procesorov, tie sa môžu následne spájať do uzlov a tvoriť tak cluster či superpočítač – teda paralelné systémy.

Podľa spôsobu zapojenia procesorov a pamätí môže byť systém typu UMA (Uniform Memory Access) alebo NUMA (non-UMA). Pri UMA systémoch je doba prístupu z procesorov do pamäti rovnaká. Pri NUMA procesor rozlišuje pamäť lokálnu a vzdialenú, logicky však tvoria súvislú pamäť. Doba prístupu do vzdialených pamäťových modulov je väčšia, čo priamo ovplyvňuje výkon. Operačný systém by to mal zohľadňovať pri alokovaní pamäte a plánovaní procesov, programátor si môže pomôcť s First Touch Policy (viac v kapitole 3.5).

V paralelných systémoch môžu byť dáta uložené okrem pamäte aj v medzipamätiach niektorých procesorov. Aby softvér fungoval správne, musia byť tieto dáta navzájom konzistentné, akoby v systéme ani medzipamäte neboli. O túto tzv. koherenciu medzipamätí sa stará hardvér procesoru. Udržiavanie medzipamätí aktualizovaných niečo stojí a pri nesprávnom použití sa môže prejaviť ako tzv. falošné zdieľanie (viď kap. 3.4.2).

2.5 Meranie výkonu

Na meranie výkonu je možné využiť hardvérové počítadlá (hardware performance counters). Ide o špeciálne registre vnútri procesorov, ktoré sú schopné zaznamenávať určité hardvérové udalosti, napr. výpadky medzipamätí, počet vykonaných inštrukcií s desatinnými číslami, počet zle predpovedaných vetvení atď. Dostupné merateľné udalosti a počet registrov (teda súčasne sledovaných udalostí) je závislý od konkrétneho procesoru.

Ich využitie je možné bez zásahu do meraného softvéru a pri každom meraní je možné zvoliť, ktoré udalosti sa majú sledovať. Pracovať s nimi je možné napr. cez rozhranie PAPI alebo nástroj PCM od Intelu.

Na meranie výkonu sa používajú dve metriky: počet prvkov za jednotku času a doba na jeden prvok. V prvom prípade sa vo vysoko náročných výpočtoch používa prevažne jednotka FLOPS - floating-point operations per second. Tá sa v tejto práci používa v kapitole 5 spolu s počtom spracovaných prvkov. V kapitole 6 sa zase meria čas strávený vo funkcii alebo v celom programe.

Základnou charakteristikou ovplyvňujúcou výkon je frekvencia procesoru. Pri dynamickom pretaktovaní sa táto frekvencia môže meniť v čase podľa spôsobu využitia procesoru. Príkladom je technológia Intel Turbo Boost, ktorá umožňuje jadrám procesoru bežať na

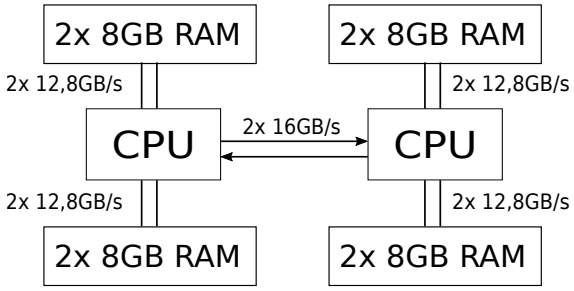
vyššej než je základná frekvencia, pokým je procesor v rámci svojich limitov (max. teplota, prúd, ...). To sa odvíja od počtu aktívnych jadier a ich záťaže. Konkrétne, vektorové inštrukcie vyžadujú viac energie a preto optimalizovaný vektorizovaný kód môže bežať na nižších frekvenciách. [10]

2.6 Anselm

Testy a merania vykonávané v tejto práci prebiehali na ostravskom clustri Anselm, ktorý vznikol v rámci projektu IT4Innovations. Teoretický výkon 94 Tflop/s zabezpečuje 209 NUMA uzlov, pričom niektoré sú vybavené GPU Kepler K20 (23) alebo MIC Xeon Phi 5110 (4). Bežné výpočtové uzly, na ktorých prebiehali testy, obsahujú 2x Intel Sandy Bridge E5-2665 (2.4 – 3.1 GHz s Turbo Boost, každý 8 jadier) a 8x 8GB DDR3 DIMM 1600Mhz. Cache sú veľkosti L1 2x32 kB (na dáta a inštrukcie), L2 256 kB a L3 veľkosti 20 MB, ktorá je zdieľaná pre 8 jadier (viď obrázok 2.2 vyššie).

Na testy sa využíval vždy 1 uzol, teda maximálne 16 jadier. Teoretický výkon jedného uzlu je 307 GFLOPS. Technológia Hyper-Threading je vypnutá a Turbo Boost je možné vypnúť pri alokovaní uzlu. Procesor podporuje inštrukcie AVX 256-bit a obsahuje 11 hardvérových počítadiel.

Operačný systém je 64b bullx Linux, ktorý je založený na Red Hat Enterprise Linux Server 6.4 a vie pracovať s NUMA systémami. [6] [4]



Obr. 2.3: Schéma výpočetného uzla Anselmu – spôsob zapojenia procesorov a pamäťových modulov spolu s priepustnosťami spojení.

2.6.1 Používanie

Na Anselm sa prihlasuje prostredníctvom služby SSH a autentizácia prebieha pomocou privátneho kľúča. Po prihlásení sa používateľ nachádza na tzv. login uzle. Ak chce počítať na superpočítači, musí si alokovať jeden alebo viac výpočetných uzlov. Uzly sú rozdelené do viacerých front. Počas tejto práce sa využívala expresná fronta `qexp` (na testovanie funkčnosti) a produkčná fronta `qprod` (na meranie výkonu a časov). Expresná fronta nebola vhodná na výkonnostné testovanie, pretože výkon spúšťaných programov sa líšil navzájom aj o 30 %, zatiaľ čo produkčné uzly dávali stabilné výsledky. [6]

Produkčné uzly boli alokované príkazom: `qsub -A OPEN-7-15 -q qprod -l select=1 -l ncpus=16 -l cpu_freq=24 -l cpu_turbo_boost=0 -I`. V príkaze, `OPEN-7-15` je identifikátor projektu, v rámci ktorého sa účtoval spotrebovaný výkon. Argument `-I` hovorí, že ide o interaktívnu reláciu, teda príkazy budú zadávané zo štandardného vstupu a nie čítané zo skriptu. Argument `-l` popisuje požiadavku na 1 uzol, 16 jadier, vypnutú technológiu TurboBoost a uzol s procesorom E5-2665 (bežiaci na frekvencii 2,4 GHz, preto `cpu_freq=24`).

Sofvér je na Anselme organizovaný do tzv. modulov. Pred použitím väčšiny softvéru je najprv nutné načítať daný modul pomocou príkazu `module load`, nasledovaným jedným alebo viacerými názvami požadovaných modulov. Dostupné moduly je možné vypísať príkazom `module avail`. V tejto práci sa využíva Intel prekladač z modulu `intel` (východzia je verzia 13), verzia 15 z modulu `intel/15.3.187`, knižnica PAPI z modulu `papi` a knižnica Intel Math Kernel Library z `mkl`. Modul prekladača stačí mať načítaný pri prekladaní, moduly `mkl` a `papi` však treba načítať aj pred používaním už preložených programov. [6]

Kapitola 3

Paralelizácia

Prvým krokom k paralelizácii programu je vytvorenie jeho behového profilu. Vychádzajúc z kapitoly 3.1 sa vyhľadajú časti kódu (napr. funkcie), v ktorých sa trávi najviac času. Programátor toho môže docieliť ručne pomocou funkcií na meranie času, s využitím nejakej knižnice alebo jednoducho použiť existujúci profilovací nástroj, ako Intel Vtune. [15]

Vo vybraných častiach kódu sa podľa kapitoly 3.2 identifikujú časti, ktoré by sa dali vykonávať paralelne. Podľa kapitoly 3.3 treba rozoznať závislosti, ktoré by mohli znemožniť efektívnu paralelizáciu. Z predchádzajúcich krokov sa odhadne možné zrýchlenie a náklady (náročnosť implementácie, potrebné prostriedky, ...). Ak by paralelizácia bola prínosom, implementuje sa podľa kapitoly 3.4. V opačnom prípade môže byť užitočné rozšíriť rozsah analýzy alebo zvážiť zmenu použitého algoritmu, či dokonca zmenu štruktúry programu.

Implementácia komunikácie medzi vláknami je popísaná v kapitole 3.4.2.

Kapitola 3.5 sa venuje zvýšeniu výkonu cez lepšie využitie medzipamätí.

Táto kapitola vychádza z [15].

3.1 Odhad zrýchlenia

Najbežnejšou aproximáciou zrýchlenia po paralelizácii je Amdahlov zákon. Ten predstavuje len odhad, pretože nepočíta s komunikáciou medzi vláknami, vplyvom vyrovnávacích pamätí či ostatnými procesmi v systéme.

$$doba\,behu = s + \frac{p}{n} \tag{3.1}$$

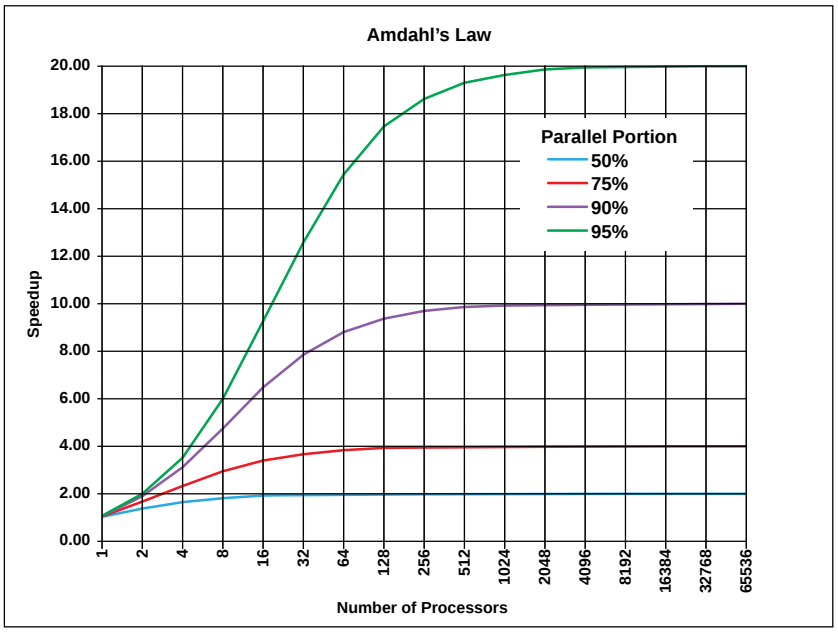
$$zrychlenie = \frac{1}{s + \frac{p}{n}} \tag{3.2}$$

Z rovnice 3.1 vyplýva, že program nemôže bežať rýchlejšie, než trvá jeho sériová časť s. Ďalej, čím viac času sa trávi v paralelizovateľnej časti p, tým väčší dopad na zrýchlenie bude mať paralelizovanie na n výpočetných jednotkách.

Z grafu 3.1 sa dá vyčítať, že ak 5 % z programu zostane sekvenčných, maximálne zrýchlenie je len 20.

3.2 Typy paralelizmu

Zjednodušene, existujú 2 kategórie paralelizmu: dátový (data parallelism) a úlohový paralelizmus (tiež funkčný, angl. task parallelism).



Obr. 3.1: Teoretické zrýchlenie programu podľa množstva paralelizovateľnej časti programu a počtu výpočetných jednotiek. [2]

Pri dátovom sa vykonávajú rovnaké operácie na rôznych dátach a v programe bývajú vo forme cyklov, napr. spracovanie prvkov vektorov a matíc. Do tejto kategórie sa zaraďuje aj vektorizácia.

Pri úlohovom paralelizme bežia rôzne úlohy súčasne. Napríklad, na úrovni operačného systému sú to jednotlivé bežiace procesy, v rámci jedného procesu to môže byť vlákno spracúvajúce vstupy používateľa a vlákno vykresľujúce rozhranie aplikácie.

3.3 Závislosti vo výpočtoch

V prípade, že s jednou premennou pracuje viac vlákien, z ktorých aspoň jedno zapisuje, jej hodnota pre vlákna závisí na poradí, v akom sa operácie vykonajú. Tabuľka 3.3 obsahuje typy závislostí, aké môžu nastať spolu s anglickými označeniami. Tie sa používajú tiež v optimalizačných výstupoch prekladača, napr. pri použití príznaku `-vec-report`.

Poradie operácií	**2. čítanie**	**2. zápis**
1. čítanie	Bez závislosti	Antizávislosť (anti)
1. zápis	Skutočná závislosť (flow)	Výstupná závislosť (output)

Tabuľka 3.1: Typy závislostí.

3.4 Paralelizácia pomocou OpenMP

OpenMP (Open MultiProcessing) predstavuje aplikačné rozhranie pre paralelizáciu a vektorizáciu pre jazyky C, C++ a Fortran. Jazyky rozširuje o špeciálne direktívy prekladača,

knižnicu funkcií a premenné prostredia, ktoré bližšie určujú, ako sa má kód paralelizovať. S minimálnou zmenou existujúcich sekvenčných programov programátor určí, ktoré časti kódu sa majú paralelizovať a o zvyšok sa postará prekladač. Práca je rozvrhovaná do vlákien, pretože oproti procesom umožňujú rýchlejšie prepínanie kontextu a vďaka zdieľanej pamäti (a teda aj medzipamäti) je komunikácia medzi nimi rýchlejšia.

Aby prekladač interpretoval OpenMP direktívy a pripojil k výslednému programu behovú knižnicu, je potrebné program prekladať s prepínačom `-openmp` (pre Intel prekladač). Pre použitie behových funkcií treba ešte použiť hlavičkový súbor `omp.h`. OpenMP bude demonštrované na jazyku C/C++ a praktické príklady sú v kapitolách 5 a 6. [9]

3.4.1 Rozdelenie práce

Prvý krokom je vytvorenie tímu vlákien pomocou direktívy `#pragma omp parallel`, za ktorou nasleduje blok kódu (tzv. paralelná oblasť), ktorý už vykonajú všetky vlákna paralelne.

Potom sa vláknam rozdelí práca. Ak sa má paralelizovať for-cyklus (s dopredu známym počtom iterácií), tak sa použije direktíva `#pragma omp for`, ktorá rozdelí iterácie rovnomerne všetkým vláknam. Ak sa jedná o cyklus s rôzne dlhými iteráciami, môže byť vhodné experimentovať so spôsobom rozvrhovania iterácií pomocou klauzuly `schedule`.

V prípade úlohového paralelizmu sa namiesto direktívy `for` použije `sections` alebo `single`.

Pre rôzne situácie sú vhodné rôzne direktívy (viď špecifikáciu OpenMP [9]), pričom ich chovanie sa dá ďalej upravovať dalšími klauzulami.

3.4.2 Súkromné a zdieľané dáta

V paralelných oblastiach kódu sa rozlišujú súkromné a zdieľané premenné. Pri súkromnej, každé vlákno má svoju vlastnú kópiu danej premennej a jej zmena v jednom vlákne nemá vplyv na ostatné vlákna. Zdieľané premenné sú spoločné pre všetky vlákna a pomocou nich prebieha komunikácia medzi vláknami. OpenMP má určité pravidlá, podľa ktorých rozoznáva súkromné a zdieľané premenné. Pomocou direktív `private` a `shared` je to možné zmeniť, čo môže v určitých prípadoch zvýšiť výkon.

Ak sa zdieľaná premenná používa na čitanie aj zápis, bude potrebné zaistenie jej správneho používania. Ak sa jedná o jednoduchšiu operáciu, ako napríklad inkrementáciu, je možné použiť direktívu `atomic`. Pre zložitejšie operácie existuje `critical`. Takáto synchronizácia so sebou nesie určitú réžiu a dochádza k serializácii vlákien, preto treba jej použitie minimalizovať.

Ak vlákna zapisujú do zdieľaných premenných, ktoré sa v pamäti nachádzajú na spoločnom riadku medzipamäti (viď kapitolu 2.3), môže nastať jav nazývaný falošné zdieľanie. Pri každom zápise jedného vlákna sa procesor snaží zabezpečiť koherenciu medzipamätí a aktualizuje aj ostatné kópie rovnakého riadku v ostatných procesoroch. To trvá nejakú dobu a preto časté zapisovanie do takýchto premenných môže znížiť výkon. Obmedziť alebo vyhnúť sa dá tomu použitím zarovnania (alignment) alebo výplne (padding) premenných, alebo zmenou algoritmu, aby a menej často používali zdieľané premenné.

3.5 Efektívnejšia práca s pamäťou

Rozdiely v dobe prístupu do pamäti v NUMA systémoch je možné zmierniť pomocou prístupu First Touch. Vtedy sa na inicializáciu dát použijú všetky vlákna, čo spôsobí, že dáta sa fyzicky umiestnia do pamäťových modulov, ktoré sú bližšie k danému vláknu. Pri výpočte sa to prejaví ako menšie čakanie na dáta z pamäti a rovnomernejšie využitie šírky pásma pamätí. Tento prístup vychádza z faktu, že moderné operačné systémy fyzicky rozmiestňujú stránky pamäte až pri ich prvom použití.

Technikou na redukovanie obmedzení zo strany pamäte je tzv. cache blocking. Ide o reorganizovanie prístupov do pamäte tak, aby sa do medzipamäti načítal blok dát, ktorý sa potom opakovanie využije a nemusel sa viackrát načítavať z pamäte. V algoritme sa to prejaví ako rozdelenie vhodných cyklov do viacerých, ktoré prechádzajú po blokoch. Veľkosť bloku je potrebné experimentálne zistiť, aby sa dosiahol najlepší výkon. [3]

Kapitola 4

Vektorizácia

Vektorové spracovanie predstavuje dátový paralelizmus na úrovní inštrukcií. Moderné prekladače podporujú vektorizáciu, pričom vygenerovaný kód bude fungovať len na procesoroch s danými SIMD inštrukciami (viď kap. 2.2). [15]

V kapitole 4.1 sa hovorí o rôznych spôsoboch ako využiť vektorové spracovanie vo vlastnom programe, od jednoduchších po náročnejšie prístupy. Kapitola 4.2 je o požiadavkach na kód, aby ho bolo možné efektívne alebo vôbec vektorizovať.

Spomínané prepínače a direktívy sa týkajú Intel prekladača a jazyka C/C++. Táto kapitola vychádza z [1].

4.1 Ako využiť vektorové spracovanie

Ak to je možné, najvhodnejšie je použiť existujúce matematické knižnice, napr. voľne dostupný ATLAS alebo Math Kernel Library (MKL) od Intelu. Tie poskytujú správny a vysoko optimalizovaný kód.

Ďalším riešením je využiť automatickú vektorizáciu prekladačom, kedy sa pri preklade vyhľadávajú určité štandardné vzory v zdrojovom kóde s dopredu známym spôsobom vektorizácie. Automatická vektorizácia sa može vykonávať buď implicitne, alebo po použití potrebných prepínačov prekladača. Ďalšími prepínačmi (napr. `-vec-report`) je možné zistiť, ktoré cykly sa vektorizovali a ktoré nie, spolu s odovôdnením. Automatická vektorizácia sa nevykonáva v prípade, ak by to mohlo spôsobiť nesprávny beh programu, alebo ak prekladač zhodnotí, že by to neprinieslo zlepšenie výkonu. V takom prípade sa dá použiť `#pragma vector always` alebo `#pragma simd`. OpenMP direktíva `#pragma omp simd` vektorizuje aj v prípade závislostí a navyše ponúka klauzuly, umožňujúce lepšiu vektorizáciu.

Pre vygenerovanie lepšieho kódu je možné poskytnúť prekladaču ďalšie informácie, a to pomocou direktív (štandardizovaných OpenMP alebo pre konkrétny prekladač), prepínačov pri preklade a kľúčových slov jazyka. Konkrétne príklady v kapitole 4.2.

Najnáročnejším prístupom je napísať dané časti v jazyku symbolických inštrukcií alebo pomocou vstavaných funkcií prekladača (angl. intrinsics).

4.2 Podmienky pre vektorizáciu

Vektorizovanie cyklov predstavuje zmenu poradia vykonávania iterácií, čo kladie určité požiadavky na cykly, viac v kapitole 4.2.1. Bez ich splnenia nemusí byť vektorizácia možná. Kapitola 4.2.2 popisuje, ako rozloženie dát v pamäti vplýva na výkonnosť ich spracovania.

4.2.1 Cykly

Aby bolo možné cyklus vektorizovať, musí byť na začiatku cyklu známy počet jeho iterácií a do konca cyklu sa nemôže meniť. To je dôvod, prečo cyklus nemôže obsahovať konštrukcie ako `break` a `return`.

Cyklus ďalej nemôže obsahovať závislosti medzi iteráciami, inak by to viedlo k nesprávnemu vysledku. Ak závislosť vzniká až po niekoľkých iteráciách a teda vektorizácia by bola možná, dá sa použiť `#pragma omp simd safelen(N)`, kde N predstavuje daný počet iterácií.

Keďže SIMD inštrukcie vykonávajú viac iterácií cyklu súčasne, nemal by cyklus obsahovať vetvenie, napr. konštrukciu `switch`. Výnimkou sú podmienené priradenia, ktoré sa dajú tzv. vymaskovať. Vtedy sa podmienená vetva počíta normálne vektorovo pre všetky iterácie, ale jej výsledky sa použijú len pre prvky, pre ktoré sa podmienka vyhodnotila ako pravdivá.

Ďalej sa treba vyhnúť volaniu funkcií. Výnimkou sú matematické funkcie vstavané v prekladači (napr. `sqrt`, `sin` a `fmax`) a funkcie, ktoré sa dajú rozvinúť v danom mieste a spĺňajú vyššie spomenuté podmienky.

4.2.2 Rozloženie dát

Pre dobrý výkon je dôležité správne zarovnanie dát v pamäti. V opačnom prípade sa musia dáta pred použitím správne zarovnávať. Premenné na zásobníku je možné zarovnať pomocou `__declspec(align(X))`, dynamicky alokovanú pamäť je potrebné získať napríklad s `_mm_malloc(N, X)`, kde X značí na koľko bajtov sa má zarovnať. Pre AVX sa odporúča zarovnanie na 32 bajtov. Ak premenné budú zarovnané, ale pri preklade to ešte nie je známe, dá sa použiť `__assume_aligned(VAR, X)` alebo `#pragma simd aligned(VAR:X)`.

Ideálna je, keď sa pristupuje k susediacim prvkom, narozdiel od prípadu, keď sú medzi prvkami rozostupy alebo ide o nepriame indexovanie prvkov. Preto je pre vektorizáciu lepšie použiť štruktúru polí a nie pole štruktúr. Tým sa redukuje počet potrebných načítaní z pamäte a tiež potrebné zarovnávanie v registroch pred výpočtom.

Potenciálnym problémom je prekrývanie ukazateľov (tzv. aliasing). Vtedy môže dochádzať k závislosti medzi dátami a vektorové spracovanie môže viesť k nesprávnemu výsledku. Ak programátor vie, že k prekrývaniu nedochádza, môže to prekladaču dať vedieť pomocou kľúčového slova `restrict`, direktívou `#pragma ivdep` alebo globálne cez prepínač pri preklade.

Kapitola 5

Minitesty

Cieľom tejto kapitoly je vyhodnotiť chovanie testovacieho počítača s využitím teórie z predchádzajúcich kapitol. Konkrétne, v kapitole 5.1 pôjde o hľadanie maximálneho prvku vektoru a v kapitole 5.2 o násobenie dvoch matíc.

Testy boli vykonávané na superpočítači Anselm, popísanom v kap 2.6 a pri testoch bola technológia Turbo Boost vypnutá. Ak nie je uvedené inak, paralelné verzie spomínané nižšie bežia na 16 vláknach. Používaný bol Intel C++ prekladač, s implicitne zapnutými optimalizáciami a vektorizáciou, tj. prepínače `-O2` a `-vec`.

Na merania bola používaná knižnica PAPI, ktorá dokáže pracovať s hardvérovými počítadlami. Aby bolo možné získať rozumné štatistiky, testy boli opakované podľa potreby (v tisícoch až miliónoch). Zmeny vysvetlené v jednej verzii sú ďalej aplikované aj v nasledujúcich verziách. Všetky zdrojové kódy sú dostupné v zložkách `vecmax` a `matmul` na priloženom CD.

5.1 Hľadanie maxima vektoru

Vo všetkých verziách ide o vyhľadanie maximálneho prvku vo vektore aj s jeho pozíciou. Vektor je tvorený pseudo-náhodnými hodnotami typu `double`.

5.1.1 Implementácia

Vo verzii 1 sa vektor prechádza v cykle for, pomocou jediného vlákna.

Verzia 2 je paralelizovaná pomocou OpenMP direktívy `parallel for`, kde sa všetky iterácie cyklu rovnomerne rozdelia medzi vlákna. Každé vlákno porovnáva svoje prvky so zdieľanou premennou `max`. Na zaistenie správnosti je potrebné použiť kritickú sekciu pomocou direktívy `critical`.

Vo verzii 3 existuje zdieľané pole pre maximá a pole pre ich pozície. Každé vlákno má tak vlastné maximum a netreba používať kritickú sekciu. Na konci sa v tomto poli maxím vyhľadá globálne maximum.

Vo 4. verzii existuje zdieľaná premenná na globálne maximum `global_max` a každé vlákno má navyše svoje privátne maximum `max`. Keď vlákno spracuje svoj interval vektoru, v kritickej sekcii porovná svoje `max` so zdieľaným `global_max` a prípadne ho prepíše. Klauzula `nowait` hovorí, že keď vlákno dokončí svoj cyklus, nemusí čakať na ostatné vlákna a môže pokračovať, v tomto prípade do kritickej sekcie.

Posledná, 5.verzia je zhodná s predchádzajúcou, líšia sa len v inicializácii dát. V 5. verzii sa využíva prístup First Touch, kde sa vektor inicializuje paralelne, pomocou direktívy

`parallel for`. Ďalej, používa sa tzv. reentrantná verzia generátoru náhodných čísel `rand_r` a každé vlákno si udržuje vlastný stav generátoru `seed`.

```
1  void vecinit(double vec[], int vec_size)
2  {
3    #pragma omp parallel
4    {
5      unsigned int seed = (unsigned int) omp_get_thread_num();
6
7      #pragma omp for
8      for (int i = 0; i < vec_size; i++)
9        vec[i] = (double) rand_r(&seed) / RAND_MAX;
10   }
11 }
```

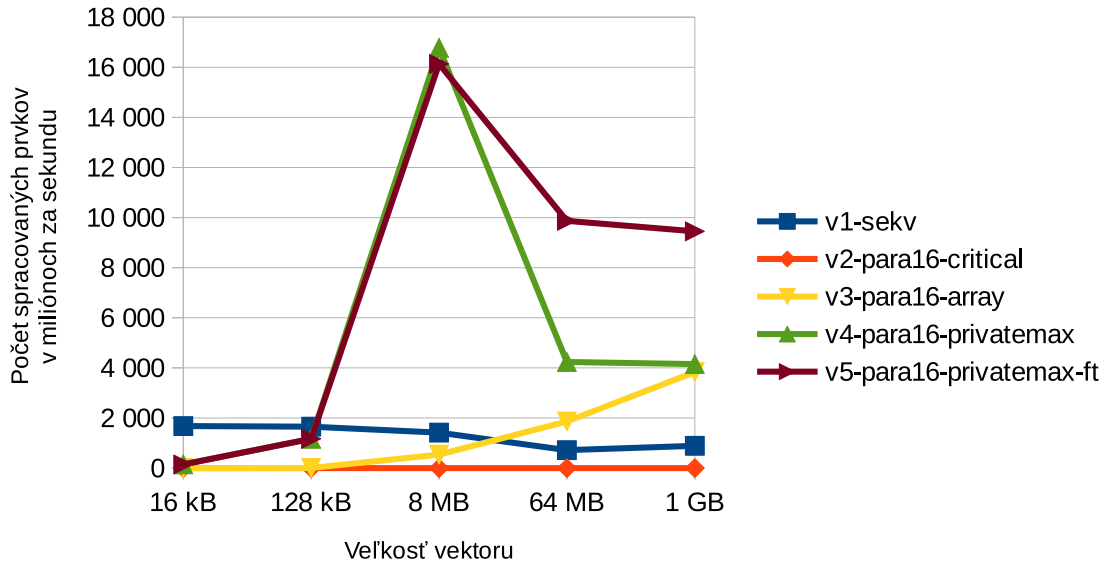
Algoritmus 5.1: First Touch inicializácia vektoru.

```
1  int global_pos = 0;
2  double global_max = -INFINITY;
3
4  // opakovanie na ziskanie rozumnych statistik
5  for (int r = 0; r < repeat; r++)
6  {
7    global_pos = 0;
8    global_max = -INFINITY;
9
10   #pragma omp parallel shared(A, data_count, global_max, global_pos)
11   {
12     int pos = 0;
13     double max = -INFINITY;
14
15     #pragma omp for schedule(static) nowait
16     for (int i = 0; i < data_count; i++)
17     {
18       if (A[i] > max)
19       {
20         max = A[i];
21         pos = i;
22       }
23     }
24
25     #pragma omp critical
26     if (max > global_max)
27     {
28       global_max = max;
29       global_pos = pos;
30     }
31   }
32 }
```

Algoritmus 5.2: Kód hľadania maxima vo vektore pre verzie 4 a 5.

5.1.2 Výsledky

Na obrázku 5.1 je možné vidieť, ako príliš časté pristupovanie ku kritickej sekcii (verzia 2) drasticky znižuje výkon. V tomto prípade, všetkých 16 vlákien pristupovalo s každým prvkom do jedinej kritickej sekcie.



Obr. 5.1: Počet spracovaných prvkov za sekundu pre rôzne verzie hľadania maxima vektoru.

Verzia 3 so zdieľaným poľom maxím, kde každé vlákno malo vlastnú položku, dopadlo len o niečo lepšie. Síce nebol potrebný atomický prístup, ale keďže jednotlivé maximá sa nachádzajú fyzicky vedľa seba, dochádza k falošnému zdieľaniu (popísané v kap. 3.5). S narastajúcou veľkosťou vektora sa výkon zlepšuje, pretože väčšie prvky sa nachádzajú stále zriedkavejšie. To predstavuje menej zápisov do poľa a teda menšiu réžiu so zabezpečením koherencie medzipamätí.

Od verzie 4 sa súkromné maximá nachádzajú na zásobníkoch jednotlivých vláken, teda nie je potrebná kritická sekcia ani nenastáva falošné zdieľanie, potom výkon rastie. Pre malé dáta (128 kB a menej) sa paralelný prístup nevyplatí, pretože réžia paralelizmu je príliš veľká. Najvyšší výkon spomedzi meraní sa dosahuje, keď sú všetky dáta (8 MB) v L3 medzipamäti (20 MB), kde ostali od inicializácie. Väčšie dáta treba potom načítavať z pomalšej operačnej pamäte a výkon klesá.

Nakoniec, v poslednej verzii sa vďaka First Touch rovnomernejšie využívajú pamäťové moduly, čo zlepšuje celkovú priepustnosť systému pre veľké objemy dát.

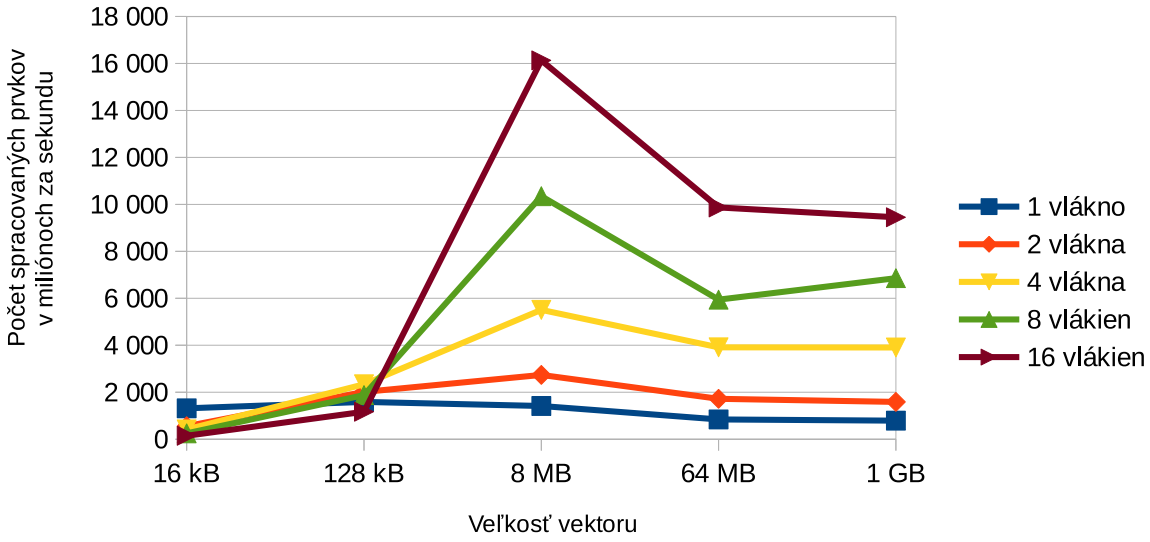
Na grafe 5.2 je tiež vidno, že pre malé dáta sa paralelizácia nevyplatí kvôli veľkej réžii. Pre veľké dáta je výkon obmedzovaný priepustnosťou pamäte.

5.2 Násobenie matíc

V tomto teste sa násobia dve štvorcové matice A a B do matice C. Matice A a B sú napĺňané hodnotami typu `double` odvodenými od indexu riadku.

5.2.1 Implementácia

Vo verzii 1 prebieha výpočet podľa klasického vzorca 5.1, na jedinom vlákne. To sa implementuje ako 3 vnorené cykly. Dáta sa alokujú štandardnou funkciou `malloc`. Táto verzia je prekladaná s vypnutými optimalizáciami, tj. s prepínačom `-O0`.



Obr. 5.2: Počet spracovaných prvkov za sekundu pre rôzne počty vlákien a veľkosti pre najrýchlejšiu verziu č. 5.

$$\mathbf{C}_{ij} = \sum_{k=1}^{m} A_{ik} B_{jk} \tag{5.1}$$

Verzia 2 má navzájom vymenené 2 vnútorné cykly. Po tejto výmene pre vnútorný cyklus platí, že maticou C a B sa prechádza s prírastkom jedného prvku a prvok z matice A zostáva konštantný. To umožňuje vektorizáciu so zachovaním správneho výsledku. Kvôli vektorizácii sú matice alokované zarovnane na 32 bajtov pomocou funkcie `_mm_malloc`. Program je prekladaný s príznakom `-xHost`, kedy prekladač použije najlepšie dostupné SIMD inštrukcie na danom počítači, teda AVX. Príznak `-fno-alias` hovorí, že v celom programe nedochádza k žiadnemu prekrývaniu ukazateľov, čo umožní lepšiu optimalizáciu.

Paralelizovaním vonkajšieho cyklu, s `#pragma omp parallel for`, vzniká verzia 3. Pre lepší výkon sa dáta inicializujú s First Touch.

Posledná verzia 4 aplikuje techniku cache blocking z kapitoly 3.5. Presnejšie, jedná sa o variantu 2D cache blocking, kedy sa matice prechádzajú najprv po 2D blokoch rozmeru `BLOCK`. Potom sa v rámci bloku prechádza po jednotlivých prvkoch. Algoritmus tak pozostáva zo 6 navzájom vnorených cyklov. Dva úplne najvnútornejšie cykly sú znovu vymenené, aby sa umožnila vektorizácia. Keďže matice sa alokujú v inej časti programu, treba prekladaču naznačiť cez `__assume_aligned(M, 32)`, že sú zarovnané. Pre zjednodušenie musí byť rozmer matíc násobkom veľkosti bloku.

Pre porovnanie bola taktiež vytvorená tzv. verzia MKL, ktorá využíva Intel Math Kernel Library pre násobenie matíc (funkcia `cblas_dgemm`). Tá vyžaduje zarovnané matice, pre lepší výkon bol použitý First Touch. Počet vlákien pre túto knižnicu sa menil pomocou OpenMP funkcie `omp_set_num_threads`.

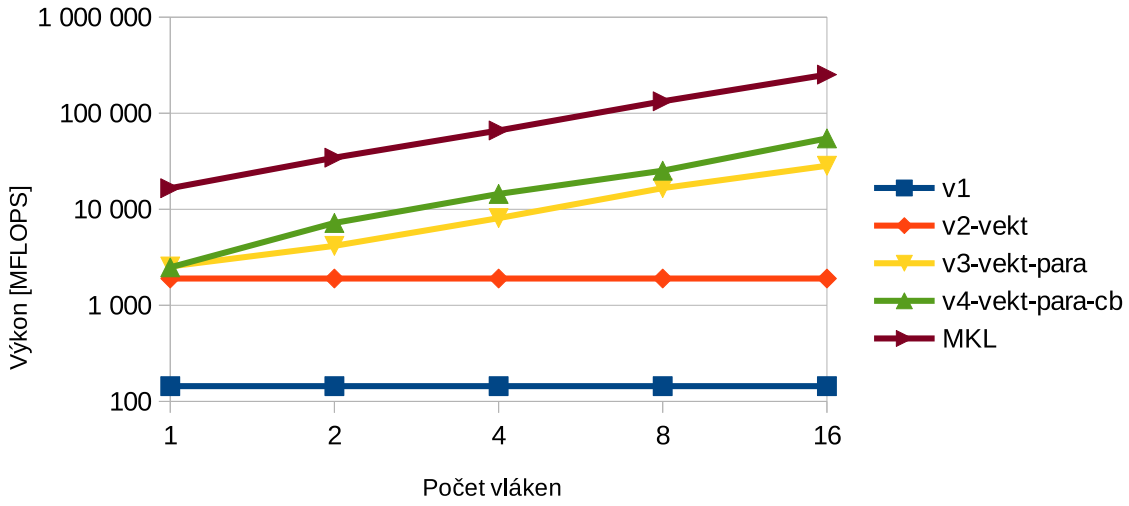
```
1  void matmul(double A[], double B[], double C[], int size)
2  {
3    __assume_aligned(A, 32);
4    __assume_aligned(B, 32);
5    __assume_aligned(C, 32);
6
7    #define BLOCK 32
8
9    #pragma omp parallel for schedule(static)
10   for (int i = 0; i < size; i++)
11     for (int j = 0; j < size; j++)
12       C[i*size+j] = 0.0;
13
14   // po blokoch
15   #pragma omp parallel for shared(A,B,C,size)
16   for (int i = 0; i < size; i += BLOCK) {
17     for (int j = 0; j < size; j += BLOCK) {
18       for (int k = 0; k < size; k += BLOCK) {
19         // v ramci bloku
20         for (int i2 = i; i2 < i+BLOCK; i2++) {
21           //prehodene k2, j2 cykly
22           for (int k2 = k; k2 < k+BLOCK; k2++) {
23             double tmp = A[i2*size+k2];
24
25             #pragma omp simd aligned(B, C: 32)
26             for (int j2 = j; j2 < j+BLOCK; j2++)
27               C[i2*size+j2] += tmp * B[k2*size+j2];
28           }
29         }
30       }
31     }
32   }
33 }
```

Algoritmus 5.3: Násobenie matíc verzie 4.

5.2.2 Výsledky

Je zrejmé, že 13násobné zrýchlenie medzi verziami 1 a 2, nie je len zásluhou vektorového spracovania (viď graf 6.1). Samotné vektorizovanie pri hodnotách s dvojitou presnosťou umožňuje maximálne 4násobné zrýchlenie. V tomto prípade sa výkon zvýšil vďaka rozdielnemu prístupu do pamäte. Vo verzii 1 sú používané prvky matice B od seba vzdialené o dĺžku celého riadku matice a preto sa musí čítať viac dát z operačnej pamäte. Verzia 2 postupuje vo vnútornom cykle s prírastkom 1, čím efektívne využíva celý riadok medzipamäte.

Na grafe 6.1 je vidno, že paralelné verzie (v3, v4, MKL) dobre škálujú s pribúdajúcim počtom vlákien. Cache blocking vo verzii 4 prináša lepšie využitie medzipamätí, čo sa odráža v takmer 2x vyššom výkone. Stále to však tvorí približne len 20 % výkonu MKL (250 GFLOPS).



Obr. 5.3: Škálovateľnosť rôznych verzií násobenia matíc o rozmeroch 2048x2048 hodnôt typu `double`.

Kapitola 6

SPH

Táto kapitola popisuje snahu paralelizovať a vektorizovať simuláciu kvapalín. Kapitola 6.1 predstavuje teoretický úvod do riešenej problematiky. Kapitola 6.2 popisuje existujúcu implementáciu, z ktorej sa vychádzalo a jej prispôsobenie pre túto prácu v 6.3. Postupujúc podľa vysvetlenej teórie sa najprv vytvorí profil aplikácie (6.4) a v kapitolách 6.5 a 6.6 sa popisuje snaha paralelizovať a vektorizovať dve najvyťaženejšie funkcie.

6.1 Úvod do SPH

Smoothed Particle Hydrodynamics (SPH) sa dá preložiť ako Simulácia časticovej hydrodynamiky s využitím vyhladzovacej funkcie. Jedná sa o výpočetnú metódu používanú na simuláciu toku kvapalín. Táto metóda vychádza z Lagrangeovho prístupu čo znamená, že kvapalina pozostáva z častíc. Oproti Eulerovmu prístupu, ktorý využíva pevnú mriežku, je síce menej presná, zato je dostatočne rýchla na použitie v reálnom čase, napr. v počítačovej grafike. Jej presnosť a rýchlosť závisí od použitej vyhladzovacej funkcie (smoothing kernel function). Pomocou nej sa vypočítava pôsobenie okolitých častíc j na časticu i, podľa ich fyzikálnych vlastností a vzájomnej vzdialenosti. V závislosti od použitej vyhladzovacej funkcie je od určitej vzdialenosti vplyv častíc zanedbateľný, čo umožňuje urýchlenie výpočtu. [18] [16]

6.2 Pôvodná implementácia

Táto kapitola vychádza z už existujúcej implementácie SPH, vytvorenej Tomom Madamsom [17]. Presnejšie, jedná sa o 2D simuláciu napísanú v jazyku C++, ktorá na vykresľovanie grafiky využíva OpenGL. V rámci jednej simulácie je možné použiť viac kvapalín (tzv. multiphase SPH), ktoré sa líšia hmotnosťou častíc a farbou. Pri jej implementácii vychádzal autor z viacerých odborných článkov [13] [18] [12].

Každá častica kvapaliny je reprezentovaná ako štruktúra, obsahujúca atribúty častice ako pozícia, hmotnosť a okolitý tlak, všetky typu `float`.

Ako bolo popísané v predchádzajúcej kapitole, častice sú ovplyvnené len časticami v ich blízkom okolí. Tento fakt sa využíva na urýchlenie výpočtu, a to rozdelením priestoru do dvojrozmernej mriežky. Potom, pri výpočte síl pôsobiacich na časticu, netreba prechádzať všetky častice ale len tie, čo sa nachádzajú v 9 okolitých bunkách mriežky. Týmto sa znižuje časová zložitosť výpočtu síl z $\mathcal{O}(n^2)$ na $\mathcal{O}(nm)$, kde n je celkový počet častíc a m je priemerný počet častíc v bunke mriežky. [18]

Každá bunka mriežky je v programe reprezentovaná ako jednosmerne viazaný zoznam častíc nachádzajúcich sa v danej bunke. Zaraďovanie častíc do mriežky podľa ich pozície v priestore prebieha vo funkcii `UpdateGrid()`.

Aplikácia beží v nekonečnej slučke podľa nasledujúceho algoritmu:

```
1  ApplyBodyForces();
2  Advance();
3  UpdateGrid();
4  CalculatePressure();
5  CalculateRelaxedPositions();
6  MoveToRelaxedPositions();
7  UpdateGrid();
8  ResolveCollisions();
9
10 RedrawScreen();
```

Algoritmus 6.1: Funkcie vykonávané v každom kroku simulácie.

6.3 Prispôsobenia pre túto prácu

Kvôli lepšiemu vyhodnocovaniu bol pôvodný zdrojový kód upravený. Počet častíc bol zväčšený z 3 000 na 100 000, čo taktiež vyžadovalo zväčšenie priestoru, v ktorom sa častice mohli pohybovať. Viac častíc vytváralo väčší tlak, čo sa prejavovalo vo forme výbuchov. Napravilo sa to zmenšením intergračného kroku na polovicu. Ďalej, všetky častice sa vytvoria hneď na začiatku v podobe dvoch obdĺžnikov, kde každý z nich bol tvorený časticami jednej kvapaliny.

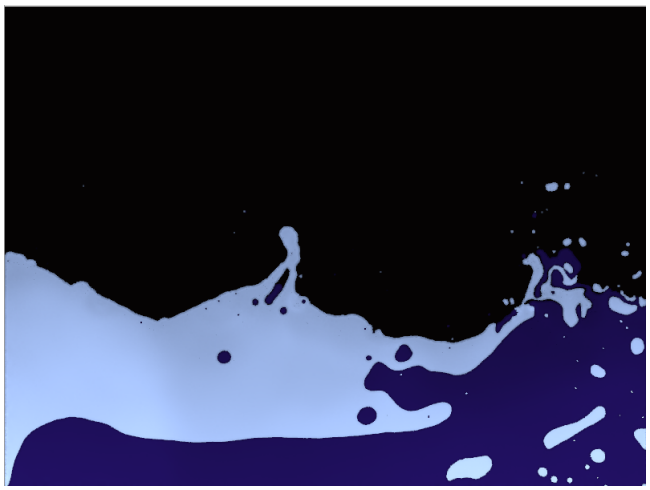
Do programu bolo zavedené obmedzenie na maximálny počet vypočítaných (vykreslených) snímok. Ak nie je spomenuté inak, tak po 500 snímkach sa program ukončil, čo sa využívalo na normalizované vyhodnocovanie výkonu.

Nakoniec, cez podmienený preklad je možné vytvoriť 3 verzie programu, podľa požadovaných výstupov. Okrem grafickej verzie pomocou OpenGL, je možné vytvoriť aj textovú verziu a verziu bez výstupov. Textová verzia vypisuje na štandardný výstup pozície všetkých častíc, a to pre každú snímku. Na merania sa používala verzia bez výstupu, pretože bol problém rozbehnúť OpenGL na Anselme.

6.4 Profil

Prvým krokom bolo vytvorenie behového profilu aplikácie, na čo sa použila knižnica PAPI. Na preklad bol používaný Intel prekladač s príznakmi `-O2` (optimalizácia pre rýchlosť) a `-xHost` (využiť najlepšie dostupné SIMD inštrukcie). Profil sa meral pre sekvenčnú verziu na jednom uzle superpočítača Anselm, s vypnutým TurboBoost. Pre porovnanie vplyvu vykresľovania bola odmeraná aj OpenGL verzia, ale na inom počítači. Oproti verzii bez výstupu trvala približne o 5 % dlhšie.

Podľa tabuľky 6.1 je vidno, že najviac času sa trávi vo funkciách `CalculatePressure` a `CalculateRelaxedPositions`. Pomocou Amdahlovho zákonu sa dá odhadnúť, či sa ich oplatí paralelizovať a aké teoretické zrýchlenie by sa malo získať, podľa vzťahu 3.2.



Obr. 6.1: Snímka z už upravenej verzie simulácie (100 000 častíc) po 3 900 krokoch.

Funkcia	**Čas [s]**	**Z celkového času**
ApplyBodyForces	0,45	0,6 %
Advance	0,53	0,7 %
UpdateGrid	1,93	2,4 %
CalculatePressure	38,64	47,5 %
CalculateRelaxedPositions	38,45	47,3 %
MoveToRelaxedPositions	0,72	0,9 %
ResolveCollisions	0,64	0,8 %

Tabuľka 6.1: Čas strávený v jednotlivý funkciách. (Poznámka: funkcia `UpdateGrid` sa v algoritme 6.1 volá na dvoch miestach.)

$$zrychlenie = \frac{1}{(1-p) + \frac{p}{n}} \tag{6.1}$$

$$zrychlenie = \frac{1}{(1-0,948) + \frac{0,948}{128}} \tag{6.2}$$

$$zrychlenie = 16,83 \tag{6.3}$$

Hodnota 128 vznikla ako 16x8, kde 16 je počet jadier jedného uzla a vektorizáciou pomocou AVX pri hodnotách typu `float` je možné spracovať 8 hodnôt súčasne v každom jadre.

Výpočet ukazuje, že paralelizovať a vektorizovať tieto dve funkcie má zmysel. Preto sa ďalšie kapitoly zameriavajú práve na ne.

Zdrojové kódy profilovanej verzie sú v adresári `sph/profile`.

6.5 Funkcia CalculatePressure

Najprv bude vysvetlené čo a ako funkcia počíta. Potom budú popísané postupy jej paralelizácie a vektorizácie, spolu s nameranými výsledkami. Zdrojové kódy týkajúce sa úprav tejto funkcie sú v adresári `sph/cp`.

6.5.1 Algoritmus

Úlohou funkcie je zistiť, ako sú častice ovplyvňované svojím okolím. Jej výsledkom je tlak v mieste každej častice, ktorý sa následne použije vo funkcii `CalculateRelaxedPositions`. Ďalším jej výstupom je zoznam najbližších susedov, a to pre každú časticu.

Ako bolo vysvetlené v časti 6.2, potenciálni susedia `pj` pre časticu `pi` sa hľadajú len v 9 najbližších bunkách mriežky (riadky 2 až 4 v algoritme 6.2). V prípade, že častica `pj` sa nachádza v dosahu vyhladzovacej funkcie, použije sa na výpočet síl a pridá sa do zoznamu susedov častice `pi` (riadky 5 až 7).

```
1 for pi in particles:
2   for cx in -1..+1:
3     for cy in -1..+1:
4       for pj in grid[pi.cellX+cx][pi.cellY+cy]:
5         if pj is in range:
6           pi.pressure += f(pj)
7           pi.neighbours.add(pj)
```

Algoritmus 6.2: Pseudoalgoritmus funkcie `CalculatePressure`.

6.5.2 Paralelizácia

Najprv je nutné zistiť, ktoré zo štyroch cyklov je možné a výhodné paralelizovať. Najvnútornejší 4. cyklus (riadok 4) prechádza cez zoznam tvorený ukazateľmi a teda nie je v tzv. kanonickej forme, ktorú požaduje OpenMP na paralelizáciu. Druhý a tretí cyklus sa vykonávajú po trikrát, resp. s použitím klauzuly `collapse(2)` deväťkrát, čo obmedzuje škálovanie na maximálne 9 vlákien. Navyše, podľa prvého riadku, tieto cykly sa vykonajú 100 000krát, čo predstavuje zvytočne veľkú réžiu paralelizácie. Najvhodnejší na paralelizovanie je prvý cyklus. Ten je v kanonickom tvare, má dostatočne veľa iterácií na využitie 16 jadier dostupných na Anselme, prípadne aj viac.

Spôsob rozvrhovania iterácií	**Čas vo funkcii [s]**
schedule(static) – východzie nastavenie	3,07
schedule(static, 10)	3,57
schedule(static, 100)	3,59
schedule(static, 1000)	3,17
schedule(dynamic, 1)	10,98
schedule(dynamic, 10)	3,61
schedule(dynamic, 100)	3,56
schedule(dynamic, 1000)	3,01

Tabuľka 6.2: Celkové trvanie funkcie `CalculatePressure` podľa použitého plánovania iterácií.

Pri paralelizácii prvého cyklu sa v ideálnom prípade bude každá častica počítať v osobitnom vlákne. Z pohľadu dátových závislostí sa čítajú premenné pre pozíciu v priestore, hmotnosť, pozíciu v mriežke a adresu častice. Zapisuje sa do premenných tlaku a zoznamu susediacich častíc. Z pohľadu prvého cyklu sa v algoritme nenachádza premenná, ktorá by sa súčasne aj čítala aj zapisovala, teda nevznikajú žiadne závislosti medzi iteráciami. Každá častica sa tak môže počítať nezávisle a v navzájom rôznom poradí. To umožňuje paralelizáciu, navyše bez nutnosti riešiť synchronizáciu vláken. Funkciu `CalculatePressure` je teda možné jednoducho paralelizovať pomocou direktívy `#pragma omp parallel for`.

Jednotlivé častice môžu mať rôzny počet susedov, pretože napr. na dne je väčší tlak a častice sú k sebe bližšie, než na hladine. Premenlivý počet susedov znamená premenlivú dĺžku výpočtu pre každú časticu. Preto môže mať zmysel experimentovať so spôsobom rozvrhovania častíc vláknam. Na to sa využije OpenMP klauzula `schedule` v konštrukcii `#pragma omp for`. Tabuľka 6.2 ukazuje celkový čas strávený vo funkcii počas simulácie, v závislosti od použitého rozvrhovania iterácií. Dynamické rozvrhovanie po 1000 časticiach je mierne rýchlejšie než východzie statické rozvrhovanie, tj. po 6250 časticiach na vlákno (100 000 / 16 vlákien). Pri dynamickom plánovaní po jednej častici je réžia priraďovania častíc vláknam príliš veľká.

Cache blocking prístup

Čím viac sa častice premiešavajú v priestore, tým viac sa výpočet spomaľuje. Ich pozícia v rámci simulácie sa mení ale v pamäti sú stále na rovnakej adrese. Pri výpočte častice je potom potrebné načítavať dáta z rôznych oblastí pamäte, čo znižuje efektivitu medzipamätí a znižuje tak výkon.

Tento neželaný jav by sa mohol zmierniť využitím cache blocking prístupu. Na to je potrebné vhodne zmeniť spôsob výpočtu. Navrhovaný spôsob je popísaný pseudoalgoritmom 6.3. Namiesto iterovania cez častice, sa môže iterovať cez bunky mriežky. Vďaka takejto organizácii prístupov do pamäte by mali častice, ktoré sú logicky pri sebe, zostať v medzipamäti a znovupoužijú sa pri výpočtoch okolitých buniek. Kvôli týmto zmenám je potrebné prispôsobiť aj výpočet síl a susedov. Tie sa potom počítajú po prírastkoch na viackrát, preto pribudol kód na ich počiatočné vynulovanie.

Spôsob rozvrhovania iterácií	Čas vo funkcii [s]
schedule(static)	8,91
schedule(dynamic, 1)	3,64
schedule(dynamic, 10)	11,36
schedule(dynamic, 100)	44,30
schedule(dynamic, 10) collapse(2)	3,47
schedule(dynamic, 100) collapse(2)	3,23

Tabuľka 6.3: Čas strávený vo funkcii `CalculatePressure` pri využití prístupu cache blocking.

```
1  # cez vsetky bunky mriezky
2  for cy in 0..gridHeight:
3    for cx in 0..gridWidth:
4
5      for pi in grid[cx][cy]:
6        reset pi.pressure, pi.neighbours
7
8      # cez 9 okolitych buniek
9      for ccy in -1..+1:
10       for ccx in -1..+1:
11
12         # pocitat prirastky pre pi od pj
13         for pi in grid[cx][cy]:
14           for pj in grid[cx+ccx][cy+ccy]:
15             if pj is in range:
16               pi.pressure += f(pj)
17               pi.neighbours.add(pj)
```

Algoritmus 6.3: Pseudoalgoritmus cache blocking prístupu vo funkcii `CalculatePressure`.

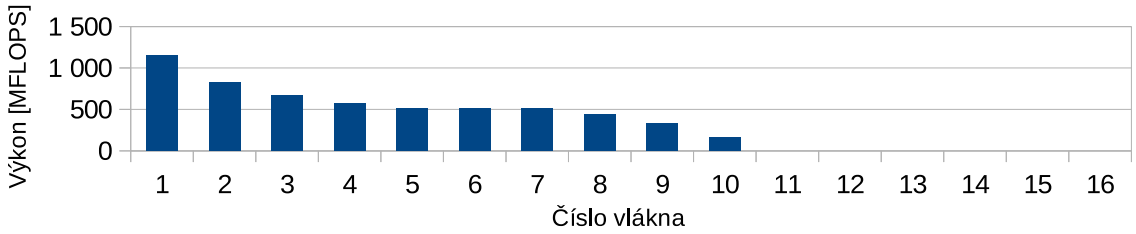
Pri tomto prístupe prvý cyklus prechádza po riadkoch mriežky, ktorých je pri použitých konštantách 125. Pri prechádzaní po riadkoch je o to dôležitejšie vybrať vhodné rozvrhovanie iterácií, pretože niektoré riadky, zvlášť horné, môžu byť úplne prázdne.

Pri statickom plánovaní je vidieť v tabuľke 6.3 dôsledok nerovnomernej distribúcie častíc v mriežke, kedy sa takmer všetky častice nachádzali v spodnej časti mriežky. Pri dynamickom plánovaní po 100 iteráciách sa práca mohla rozdeliť len do 2 vlákien. Dynamicky po 10 iteráciách sa dokázalo využiť 13 vlákien, avšak tie boli zase nerovnomerne využité kvôli umiestneniu častíc v mriežke. V tomto prípade bolo možné aplikovať klauzulu `collapse(2)`, ktorá umožnila rozvrhovať iterácie z dvoch najbližších cyklov, teda dokopy 20 750 iterácií (125x166). Lepší výsledok potom dosiahlo priraďovanie po 100 bunkách.

Táto implementácia cache blockingu bola teda oproti pôvodnej verzii asi o 7 % pomalšia. To však platí pre daný spôsob vytvárania častíc (v tvare 2 obdĺžnikov blízko dna) a len 500 krokov simulácie. Pri dynamickejších scénach, napr. na spôsob fontány alebo s prekážkami, a po viac krokoch, by boli častice viac premiešané. V takých prípadoch by táto verzia mohla podávať lepšie výkony v porovnaní s pôvodnou.

6.5.3 Vektorizácia

Vektorizácia tejto funkcie nie je taká jednoduchá a priamočiara ako paralelizácia. Pri pohľade na skutočný kód 6.4 sa vynorí niekoľko prekážok. Hneď prvou je typ cyklu. Ten



Obr. 6.2: Výkon jednotlivých vlákien pri statickom plánovaní. Je vidieť, že prvé vlákno počítalo spodné riadky mriežky.

prechádza zoznamom častíc tvoreným ukazateľmi a preto nie je možné určiť celkový počet iterácií. To je dôvod, prečo po preklade s príznakom `-vec-report` prekladač hlási *„non-standard loop is not a vectorization candidate“*. Podľa teórie je ďalšou prekážkou príkaz `continue`, ktorý sa nedá vektorizovať. Ďalej, premenné `density` a `nearDensity` sa v každej iterácii čítajú aj zapisujú, čo predstavuje cyklom prenášanú závislosť. Prekladač ešte hlási závislosť premennej `count`.

```
1  for (Particle* ppj=grid[ni+nj]; NULL!=ppj; ppj=ppj->next)
2  {
3      const Particle& pj = *ppj;
4
5      float dx = pj.x - pi.x;
6      float dy = pj.y - pi.y;
7      float r2 = dx*dx + dy*dy;
8      if (r2 < kEpsilon2 || r2 > kH*kH)
9          continue;
10
11     float r = sqrt(r2);
12     float a = 1 - r/kH;
13     density += pj.m * a*a*a * kNorm;
14     nearDensity += pj.m * a*a*a*a * kNearNorm;
15
16     if (neighbours[i].count < kMaxNeighbourCount)
17     {
18         neighbours[i].particles[neighbours[i].count] = &pj;
19         neighbours[i].r[neighbours[i].count] = r;
20         ++neighbours[i].count;
21     }
22 }
```

Algoritmus 6.4: Reálny kód najvnútornejšieho cyklu.

Prvého problému - prechádzanie po ukazateľoch - by sa dalo zbaviť tak, že by sa celý zoznam najprv nakopíroval do (zarovnanej) štruktúry polí a pri tom by sa zistila aj jeho dĺžka. Potom by sa prechádzalo cez tieto polia.

Možno to tak na prvý pohľad nevypadá, ale príkaz `continue` sa dá transformovať na podmienené priradenie premenných `density` a `nearDensity`, čo sa už dá vektorovo vymaskovať. Je to možné preto, že vyhodnotenie prírastkov týchto premenných je možné nezávisle od platnosti podmienky, ale za cenu zbytočných výpočtov. To je aj dôvod, prečo to prekladač nehlási ako problém.

Aj napriek tomu, že sa `density` a `nearDensity` v každej iterácii čítajú aj zapisujú, vektorizácia je možná. Moderné prekladače dokážu tento vzor tzv. redukcie rozpoznať a

vektorizovať. [11]

Poslednou prekážkou vo vektorizácii je závislosť premennej `count`. Tento problém sa dá vyriešiť vyňatím celej podmienky mimo cyklu. Tá sa potom vykoná v osobitnom cykle pre všetky častice v poli. To ďalej so sebou nesie ďalšie úpravy, aby bolo možné rozoznať, ktoré častice treba zaradiť do zoznamu susedov a ktoré nie.

Po všetkých týchto úpravách a pridaní `#pragma vector always` sa síce funkcia nakoniec vektorizovala, ale kvôli pridanej réžii sa spomalila z 3,03 na 4,83 sekundy.

Vektorizovanie cache blocking verzie

V predchádzajúcom prípade sa muselo pre každú časticu kopírovať 9 zoznamov častíc do štruktúry polí, čo spôsobilo dané spomalenie. Pri vektorizovaní cache blocking verzie z kapitoly 6.5.2 by sa malo toto kopírovanie redukovať.

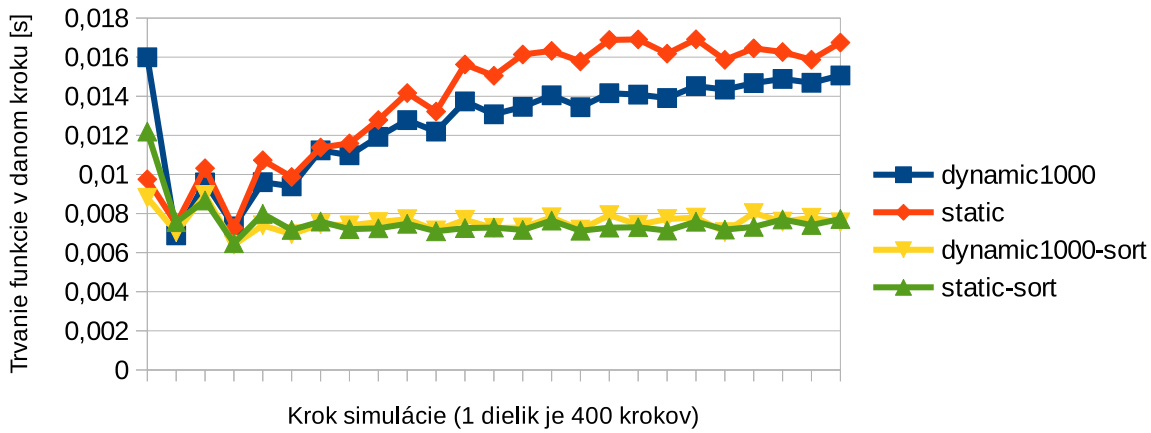
V tomto prípade sa najprv nakopíruje (počítaná) bunka i do štruktúry polí. Následne sa kopíruje každá z 9 okolitých (čítaných) buniek j. Bunka i by sa mala použiť 9x a zlepšiť tak efekt medzipamätí. Bunka j musí byť v poli, aby sa umožnila vektorizácia. Kvôli vektorizácii sú všetky tieto polia zarovnané na 32 bajtov.

Síce bol tento spôsob vektorizovania rýchlejší (3,65 sekundy), stále je verzia bez vektorizácie rýchlejšia. Spôsobené to bolo pravdepodobne pridanou réžiou.

6.5.4 Postupné spomaľovanie výpočtu

Ako bolo spomínané vyššie (6.5.2), premiešavaním častíc sa výpočet postupne spomaľoval. Tento jav sa dá redukovať občasným zoradením častíc, aby tak ich fyzické umiestnenie v pamäti viac zodpovedalo umiestneniu v priestore simulácie.

Zoraďovanie častíc bolo pridané do funkcie `UpdateGrid`. Je implementované pomocou C++ funkcie `sort` a zoraďuje sa podľa indexu v mriežke, ktorý sa získa transformáciou súradníc častice. Tímto sa dosiahne, že častice nachádzajúce sa v jednej bunke mriežky, sú uložené v pamäti blízko seba. Zoraďovanie sa vykonáva každých 500 snímok a preto na vyhodnocovanie jeho dopadu na výkon bolo treba predĺžiť simuláciu, konkrétne na 10 000 snímok/krokov. Časy boli merané pomocou OpenMP funkcie `omp_get_wtime`. Pridanie zoraďovania si vyžiadalo aj úpravy v spracovaní materiálu častíc pri ich vykresľovaní.



Obr. 6.3: Aktuálne trvanie funkcie `CalculatePressure` v danom kroku.

Spôsob rozvrhovania iterácií	**Čas vo funkcii [s]**
schedule(static) – východzie nastavenie	3,00
schedule(static, 1)	3,04
schedule(static, 10)	2,66
schedule(static, 100)	2,63
schedule(static, 1000)	2,99
schedule(dynamic, 1)	6,14
schedule(dynamic, 10)	2,70
schedule(dynamic, 100)	2,59
schedule(dynamic, 1000)	2,79

Tabuľka 6.4: Celkové trvanie funkcie `CalculateRelaxedPositions` pri rôznom spôsobe plánovania.

Na grafe 6.3 je vidno, že ak sa častice nezoraďujú, tak je z dlhodobého hľadiska lepšie použiť dynamické plánovanie po 1000 časticiach/iteráciách oproti východziemu statickému plánovaniu. Vďaka zoraďovaniu je však možné zrýchliť funkciu `CalculatePressure` až na polovicu. Pri OpenGL verzii je pri tom zoradenie častíc veľným okom nepostrehnuteľné.

6.6 Funkcia CalculateRelaxedPositions

Zdrojové súbory venujúce sa tejto funkcii sú v adresári `sph/crp`.

6.6.1 Algoritmus

V tejto funkcii sa využije vypočítaný tlak v mieste častice a s ohľadom na vlastnosti ako povrchové napätie a viskozita kvapaliny sa vypočíta nová pozícia častice v priestore.

```
1 for pi in particles:
2   for pj in pi.neighbours:
3     pi.position += f(pi, pj)
4
5     if pi.mass == pj.mass:
6       pi.position += g(pi, pj)
7
8     if u(pi,pj) > 0:
9       pi.position += v(pj)
```

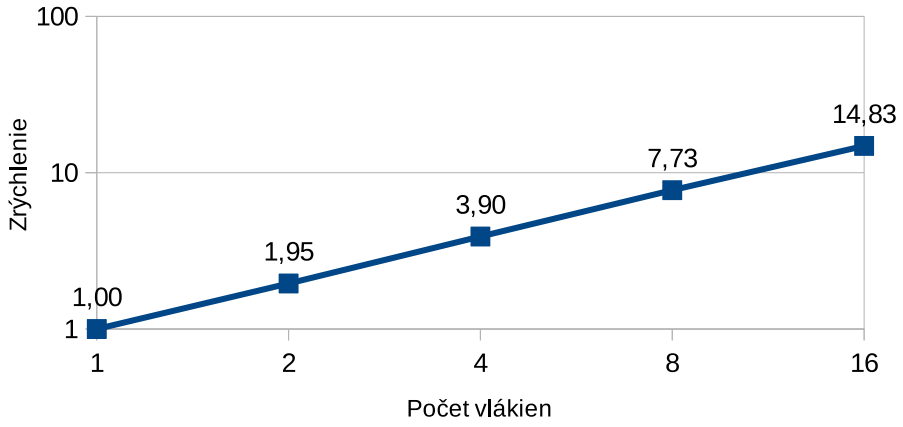
Algoritmus 6.5: Pseudoalgoritmus funkcie `CalculateRelaxedPositions`.

6.6.2 Paralelizácia

Vhodným kandidátom na paralelizáciu je prvý cyklus. Počet iterácií je známy na začiatku cyklu a algoritmus neobsahuje žiadne cyklom prenášané závislosti. Preto je ho možné paralelizovať pomocou `#pragma omp parallel for`. Taktiež je možné experimentovať s rôznymi spôsobmi plánovania iterácií. Podľa tabuľky 6.4 vychádza dynamické a statické plánovanie po 100 časticiach najrýchlejšie.

Iný spôsob výpočtu, ako v prípade funkcie `CalculatePressure`, asi nie je možný.

Na grafe 6.4 je ukázané, ako funkcia veľmi dobre škáluje s pribúdajúcim počtom vlákien. Počet vlákien sa menil cez premennú prostriedia `OMP_NUM_THREADS`.



Obr. 6.4: Zrýchlenie oproti jednému vláknu.

6.6.3 Vektorizácia

Analýzou vnútorného cyklu z pohľadu vektorizácie sa dá odhaliť cyklom prenášaná závislosť premennej uchovávajúcej pozíciu častice (v skutočnosti 2 premenné – `x` a `y`). Ďalší problém môžu predstavovať 2 podmienky.

Pri vektorizácii tejto funkcie boli testované 2 typy direktív – OpenMP a direktívy Intel prekladača.

Pri použití `#pragma omp simd` a `#pragma vector always` prekladač nevedel kód vektorizovať z dôvodu závislosti premenných `x` a `y`. V prípade OpenMP direktívy pomohlo pripísanie klauzuly `reduction(+:x,y)`. Síce Intel direktíva nedokáže v tomto prípade vektorizovať 3 redukcie v rámci jednej iterácie, pri `CalculatePressure` dokázala vektorizovať jednu redukciu. Preto sa najprv vypočítajú prírastky `x` a `y` do lokálnych premenných, tie sa potom pred koncom iterácie zredukujú. Tým sa umožní vektorizácia.

Obidve podmienky dokáže prekladač vymaskovať, bez akýchkoľvek potrebných úprav.

Ďalším pokusom urýchliť výpočet bolo vyňať spoločnú časť výpočtov, aby sa počítala len raz (v tabuľke 6.5 označené ako common).

Ďalej, meraniami bolo zistené, že podmienka na riadku 8 sa takmer vždy vyhodnotí ako pravdivá. Zároveň, vykonanie kódu na riadku 9 aj v prípade neplatnosti podmienky nemá vplyv na funkčnosť algoritmu a pravdepodobne má ísť len o ušetrenie času. V skutočnosti však práve zbytočné vyhodnocovanie podmienky môže spomaľovať kód (súvisí so zreťazených spracovaním, viď kapitola 2.1).

V tabuľke 6.5 vidno, že nevyhodnocovanie podmienky na riadku 8 nemá takmer žiadny vplyv na rýchlosť výpočtu. Vzhľadom na to, že je takmer vždy platná, procesor dokáže jej hodnotu správne predvídať. Snaha predpočítať spoločnú časť výpočtu tiež nemá významnejší vplyv.

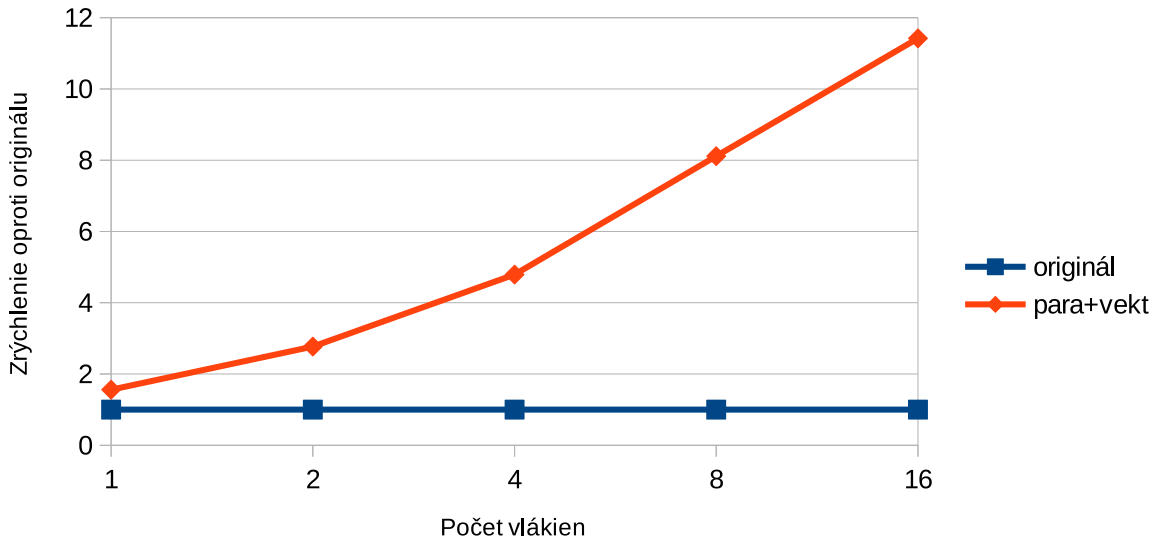
Tieto merania (a tiež v kapitole 6.5.3) boli prekladané pomocou Intel prekladača verzie 15. Tá, narozdiel od východzej verzie 13 na Anselme, podporuje vektorovú redukciu a OpenMP vektorizáciu vôbec. [7]

Vlastnosti	Vektorizované	Čas [s]
omp simd	nie	2,60
omp simd reduction	áno	1,66
omp simd reduction, bez podmienky	áno	1,62
vector always	nie	2,60
vector always, lokálne prírastky	áno	1,61
vector always, lokálne prírastky	áno	1,58
vector always, lokálne prírastky, bez podmienky	áno	1,59

Tabuľka 6.5: Celkové trvanie funkcie `CalculateRelaxedPositions` v závislosti od použitého spôsobu vektorizácie.

6.7 Najlepšia verzia

Najrýchlejšia verzia vznikla paralelizáciou funkcie `CalculatePressure` použitím dynamického prideľovania iterácií po tisíc a funkcie `CalculateRelaxedPositions` tiež dynamicky, po 100 iteráciách. Na zrýchlení sa podieľa aj zoraďovanie častíc v pravidelných intervaloch. Druhá spomínaná funkcia bola vektorizovaná direktívou Intel prekladača `#pragma vector always`, čo si ďalej vyžiadalo použitie lokálnych premenných na prírastky. Malé zlepšie prinieslo predpočítanie spoločnej časti dvoch výpočtov. Aby bola takáto vektorizácia možná, bolo potrebné použiť Intel prekladač verzie 15. Výsledná verzia bola porovnávaná s upravenou variantou pôvodného algoritmu. Simulácia bežala po dobu 4 000 krokov, čo v prípade pôvodného algoritmu trvalo viac než 15 minút.



Obr. 6.5: Zrýchlenie finálnej verzie oproti upravenej pôvodnej.

V grafe 6.5 je vidno, ako aplikácia škáluje s pribúdajúcim počtom vlákien. Síce sa nejedná o ideálne škálovanie (ako v prípade násobenia matíc), zväčšovanie počtu vlákien prináša citeľné zrýchlenie. Pri použití 16 vlákien to predstavuje zrýchlenie 11,4x oproti pôvodnej verzii.

Čistá vektorizácia je rozoznateľná pri použití jedného vlákna, kedy sa výpočet zrýchlil 1,56krát.

Teoretické zrýchlenie 16,83 sa nedosiahlo preto, že algoritmy sú závislé na využívaní zoznamov ukazateľov. To je presný opak toho, čo sa požaduje pre vektorizáciu (súvislé dáta v pamäti) a kvôli nepredvídateľnému prístupu do pamäte je potrebné čakať na načítanie dát.

Výsledná verzia bola prekladaná aj s príznakom `-O3` (agresívna optimalizácia na rýchlosť) a `-ipo` (Interprocedural Optimization). Prípadné zrýchlenie bolo zanedbateľné. Ako sa píše vo vysvetlivkách k prekladaču, tieto optimalizácie nemusia zlepšiť výkon pre niektoré programy. Ďalej bolo testované, aký vplyv má zmenšenie intervalu medzi zoradeniami častíc (z 500 na 250 krokov). To však nemalo výraznejší vplyv na dobu behu.

Súbory k tejto verzii sa nachádzajú v adresári `sph/final`.

Kapitola 7

Záver

V tejto práci som staval na existujúcom kóde simulácie toku kvapalín a paralelizoval a vektorizoval som ho pomocou OpenMP a Intel prekladača. Vyskúšal som pri tom viaceré prístupy, ako cache blocking, zoraďovanie dát počas behu či dočasné reorganizovanie dát. Z nich som vybral tie najlepšie a vytvoril verziu 11,4x rýchlejšiu než bola pôvodná.

V tomto prípade sa ukázalo, že tie jednoduchšie riešenia sú efektívnejšie. Napriek tomu, že snahou bolo maximalizovať rýchlosť programu na danom stroji, vďaka použitiu OpenMP bude aplikácia dobre škálovať aj na iných strojoch, s inými počtami jadier.

Potenciál vektorových jednotiek však zostal takmer nevyužitý. Môžu za to použité zoznamy ukazateľov, ktoré sú pre vektorizáciu nevhodné. Namiesto nich by mohla pomôcť štruktúra obsahujúca celé častice. To však so sebou nesie ďalšie komplikácie, ako zvýšená spotreba pamäte a potreba prispôsobiť aj ďalšie štruktúry a algoritmy v programe. Ďalším riešením by mohla byť úplná zmena štruktúry aplikácie tak, aby bola priateľskejšia voči vektorovému spracovaniu.

Nové zistenia o chovaní aplikácie a možné riešenia by mohlo priniesť otestovanie aplikácie na nedávno vytvorenom ostravskom superpočítači Salomon. Ten disponuje az 24 jadrami na uzol a podporuje AVX2 s inštrukciami FMA. Tie by mohli zvýšiť výkon aplikácie bez zásahu do kódu. Okrem lepšieho hardvéru, na Salomone sú aj novšie verzie softvéru než na Anselme, napr. Intel Advisor 2016 špecializujúci sa na vektorizáciu.

Literatúra

[1] A Guide to Auto-vectorization with Intel C++ Compilers. [Online; navštívené 3.5.2016].
URL https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers

[2] Amdahl's law. [Online; navštívené 16.5.2016].
URL https://en.wikipedia.org/wiki/Amdahl's_law

[3] Cache Blocking Techniques. [Online; navštívené 10.2.2015].
URL https://software.intel.com/en-us/articles/cache-blocking-techniques

[4] Enhancements to NUMA in Red Hat Enterprise Linux 6. [Online; navštívené 21.4.2016].
URL https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-cpu-numa-enhancements.html

[5] Haswell's Wide Execution Engine. [Online; navštívené 2.5.2016].
URL http://www.anandtech.com/show/6355/intels-haswell-architecture/8

[6] https://docs.it4i.cz/anselm-cluster-documentation. [Online; navštívené 21.1.2015].
URL Anselmclusterdocumentation-Introduction

[7] Intel C++ Compiler - ANSI C/C++ and OpenMP* compliance. [Online; navštívené 11.5.2016].
URL https://software.intel.com/en-us/articles/intel-c-compiler-ansi-cc-compliance

[8] NEON - ARM. [Online; navštívené 17.4.2016].
URL http://www.arm.com/products/processors/technologies/neon.php

[9] OpenMP Application Program Interface. [Online; navštívené 18.10.2015].
URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[10] Optimizing Performance with Intel Advanced Vector Extensions. [Online; navštívené 2.12.2015].
URL http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf

[11] simd. [Online; navštívené 14.5.2016].
URL https://software.intel.com/en-us/node/524555

[12] Becker, M.; Teschner, M.: Weakly Compressible SPH for Free Surface Flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, ISBN 978-1-59593-624-0, s. 209–217.
URL http://dl.acm.org/citation.cfm?id=1272690.1272719

[13] Clavet, S.; Beaudoin, P.; Poulin, P.: Particle-based Viscoelastic Fluid Simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-198-8, s. 219–228, doi:10.1145/1073368.1073400.
URL http://doi.acm.org/10.1145/1073368.1073400

[14] Eggers, S. J.; Emer, J. S.; Levy, H. M.; aj.: Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, ročník 17, č. 5, Sept 1997: s. 12–19, ISSN 0272-1732, doi:10.1109/40.621209.

[15] Gove, D.: *Programování aplikací pro vícejádrové procesory*. Computer Press, a.s., 2011, ISBN 978-80-251-3487-0.

[16] Horanský, M.: Simulácia časticovej hydrodynamiky v zložitých útvaroch s využitím vyhladzovacej funkcie. [Online; navštívené 12.5.2016].
URL http://dai.fmph.uniba.sk/upload/2/2c/Horansky_P12.pdf

[17] Madams, T.: Why my fluids don't flow. [Online; navštívené 22.11.2015].
URL https://imdoingitwrong.wordpress.com/tag/smoothed-particle-hydrodynamics/

[18] Müller, M.; Charypar, D.; Gross, M.: Particle-based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, ISBN 1-58113-659-5, s. 154–159.
URL http://dl.acm.org/citation.cfm?id=846276.846298

[19] Pacheco, P. S.: *An introduction to parallel programming*. Elsevier Inv., 2011, ISBN 978-0-12-374260-5.

Přílohy

Seznam příloh