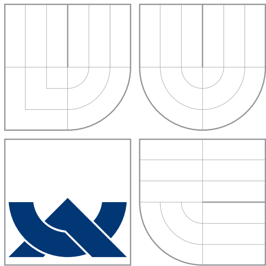
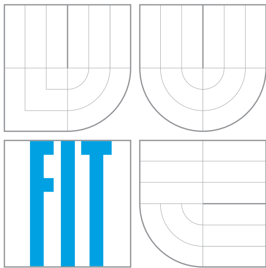


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ASISTOVANÁ VEKTORIZACE A PARALELIZACE KÓDU POMOCÍ STANDARDU OPENMP 4.0

ASSISTED CODE VECTORIZATION AND PARALLELIZATION USING THE OPENMP 4.0 STANDARD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ SLOUKA
AUTHOR

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

BRNO 2016

Zadání bakalářské práce/17433/2015/xslouk02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů | Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Slouka Lukáš**

Obor: Informační technologie

Téma: **Asistovaná vektorizace a paralelizace kódu pomocí standardu OpenMP 4.0**

Assisted Code Vectorization and Parallelization Using the OpenMP 4.0 Standard

Kategorie: Počítačová architektura

Pokyny:

1. Seznamte se s architekturou moderních vícejádrových procesorů. Zaměřte se především na organizaci vyrovnávacích pamětí a vektorová rozšíření typu AVX.
2. Seznamte se s prostředky pro asistovanou vektorizaci a paralelizaci kódu pomocí standardu OpenMP 4.0.
3. Navrhněte a implementujte sadu mikrotestů s cílem vyhodnotit základní parametry použitých procesorů a osvojit si standard OpenMP 4.0.
4. Zvolte vhodný problém na kterém budete demonstrovat výhody asistované vektorizace a paralelizace vůči automatické a manuální variantě.
5. Implementuje a optimalizujte vybraný problém na superpočítačích Anselm a Salomon.
6. Porovnejte výkonnost navržených implementací a zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D.**, UPSY FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.

vedoucí ústavu

Abstrakt
Predmetom bakalárskej práce je optimalizácia kódu pomocou štandardu OpenMP 4.0, ktorý poskytuje prostriedky pre asistovanú paralelizáciu a vektorizáciu. Okrem popisu štandardu OpenMP 4.0 práca obsahuje aj náhľad do architektúr moderných počítačov a to najmä systému rýchlych vyrovnávacích pamätí a modulov SSE/AVX, ktoré hrajú veľmi významnú rolu v oblasti optimalizácie. Práca demonštruje výhody optimalizovaného kódu pomocou štandardu OpenMP 4.0 oproti neoptimalozavanému kódu na sade benchmarkov zameraných na rôzne aspekty optimalizácie.

Abstract
The subject of the bachelor's thesis is code optimization using the OpenMP 4.0 standard which provides tools for assisted parallelization and vectorization. In addition to the description of the OpenMP 4.0 standard, the thesis as well contains an insight into architectures of modern computers, specifically the system of cache memories and SSE/AVX modules that play a major role in the optimization field. The thesis demonstrates advantages of optimized code compared to unoptimized version on a set of benchmarks which are aimed at various aspects of optimization.

Klíčová slova
OpenMP 4.0, optimalizácia, vektorizácia, paralelizácia, vyrovnávacia pamäť, SSE, AVX, výkonnost, PAPI, VTune, benchmark, skalárny súčin vektorov, binárne vyhľadávanie, mergesort, maticový súčin, numerické riešenie

Keywords
OpenMP 4.0, optimization, vectorization, parallelization, cache, SSE, AVX, performance, PAPI, VTune, benchmark, vector dot product, binary seach, mergesort, matrix product, numerical solution

Citace
Lukáš Slouka: Asistovaná vektorizace a paralelizace kódu pomocí standardu OpenMP 4.0, bakalářská práce, Brno, FIT VUT v Brně, 2016

Asistovaná vektorizace a paralelizace kódu pomocí standardu OpenMP 4.0

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 Slouka
17. května 2016

Poděkování

Rád by som sa poďakoval vedúcemu práce Ing. Jiřímu Jarošovi za jeho neoceniteľnú pomoc pri tvorbe tejto práce.

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „IT4Innovations National Supercomputing Center – LM2015070“.

© Lukáš Slouka, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teoretická časť **4**
- 2.1 Architektúry moderných procesorov . . . 4
  - 2.1.1 Taxonómia paralelných počítačov . . . 4
  - 2.1.2 Rýchla vyrovnávacia pamäť (cache) . . . 5
  - 2.1.3 Vektorové rozšírenia inštrukčnej sady . . . 6
- 2.2 Supercomputing . . . 7
  - 2.2.1 Salomon . . . 7
  - 2.2.2 Anselm . . . 7
- 2.3 Optimalizácia kódu . . . 7
- 2.4 OpenMP 4.0 . . . 8
  - 2.4.1 Programovací model . . . 9
  - 2.4.2 Runtime funkcie . . . 9
  - 2.4.3 Premenné prostredia . . . 9
  - 2.4.4 Synchronizácia . . . 10
  - 2.4.5 Deľba práce . . . 10
  - 2.4.6 Dátový rámec . . . 11
  - 2.4.7 OpenMP úlohy . . . 11
  - 2.4.8 Asistovaná vektorizácia . . . 12
- 2.5 Analýza výkonnosti . . . 13
  - 2.5.1 Metriky . . . 13
  - 2.5.2 Nástroje . . . 14

3 Praktická časť **16**
- 3.1 Automatická a asistovaná vektorizácia . . . 16
- 3.2 Jednoduché operácie nad vektormi . . . 18
  - 3.2.1 Násobenie konštantou . . . 18
  - 3.2.2 Skalárny súčin . . . 20
- 3.3 Mergesort . . . 21
- 3.4 Binárne vyhľadávanie . . . 25
- 3.5 Maticový súčin . . . 27
  - 3.5.1 Naivný algoritmus . . . 27
  - 3.5.2 Algoritmus s využitím transponovanej matice . . . 29
  - 3.5.3 Blokový algoritmus . . . 30
- 3.6 Numerické riešenie LaPlaceovej rovnice . . . 34
  - 3.6.1 Jacobiho metóda . . . 35
  - 3.6.2 Gauss-Seidelova metóda . . . 36

4 Záver 39

Literatúra 41

Prílohy 43
Zoznam príloh 44

A Obsah CD 45

B Kompletný zoznam OpenMP 4.0 runtime funkcií 46

C Kompletný zoznam OpenMP 4.0 premenných prostredia 48

D PBS Pro 49

E Zoznam dôležitých optimalizačných prepínačov Intel C/C++ prekladača 50

F Implementácia blokového algoritmu maticového súčinu 52

G Maticový súčin - cache 53

H Odvodenie vzorca pre numerický výpočet LaPlaceovej rovnice 55

Kapitola 1

Úvod

Výkonnosť počítačových systémov je jedným z najdôležitejších faktorov reflektujúcich nie len ich kvalitu ale aj cenu na trhu. Z pohľadu hardware je výkonnosť priamo úmerná počtu tranzistorov na čipe procesora. V roku 1965 spoluzakladateľ firmy Intel Gordon Moore publikoval myšlienku, že množstvo tranzistorov, ktoré môžu byť umiestnené na integrovanom obvode sa pri zachovaní rovnakej ceny približne každých 18 mesiacov zdvojnásobí. Táto myšlienka je bežne známa ako Moorov zákon [21]. Je ohromujúce, že po vyše 40 rokoch tento zákon stále platí. Zaujímavým výsledkom tohto trendu je, že ľudia vo všeobecnosti považovali hardware za jediný zdroj výkonnosti. To viedlo k vytvoreniu neefektívnych programovacích jazykov, ktoré dodnes neposkytujú dostatočnú podporu pre softwarovú optimalizáciu.

S narastajúcim počtom tranzistorov sa však zvyšovala aj spotreba energie a to takmer kvadraticky [2]. Tento ďalej neúnosný trend spôsobil výrazný zlom v spôsobe konštrukcie procesorov, kedy konštruktéri začali modelovať procesory s ohľadom na optimalizáciu spotreby energie a nie výkonnosti bez ohľadu na spotrebu. Výsledkom bola nová architektúra ktorá namiesto jedného jadra rozdelila procesor na viac paralelných jadier bežiacich na nižšej frekvencii. Moderné procesory poskytujú softwarovým vývojárom okrem viacerých jadier aj rozšírenia umožňujúce vektorové spracovanie dát.

Tieto zmeny viedli ku zvyšovaniu výkonnosti pomocou paralelného a vektorového programovania a teda aj k vytváraniu nástrojov pre podporu paralelizácie a vektorizácie ako je štandard OpenMP 4.0 [18].

Cieľom tejto práce je ukázať dôležitosť optimalizácie kódu v aplikáciach určených pre moderné procesory. Jadro práce je rozdelené na teoretickú a praktickú časť. Teoretická časť je venovaná súčasným architektúram procesorov a princípom, ktoré je nutné poznať a pochopiť pre efektívnu implementáciu paralelných programov. Ďalej obsahuje základnú špecifikáciu štandardu OpenMP 4.0 a taktiež sa venuje spôsobom merania a analýzy výkonnosti procesorov a moderným nástrojom určeným na túto činnosť. V praktickej časti sú všetky predchádzajúce koncepty ukázané na sade benchmarkov.

Kapitola 2

Teoretická časť

2.1 Architektúry moderných procesorov

V priebehu posledných 40 rokov bol technologický pokrok v oblasti digitálnych počítačov jedným z najväčších úspechov vedy ako takej. Vďaka tomuto pokroku bolo možné umiestňovať čoraz viac tranzistorov na integrované obvody a pracovať na čoraz vyšších frekvenciách. Výsledkom tohto hardwarového zvyšovania výkonnosti bola vysoká spotreba energie a nadmerné generovanie tepla. Dizajnéri čipov reagovali znižovaním operačného napätia. Toto riešenie však nebolo dostačujúce a jedinou alternatívou bolo umiestniť na čip procesora viac výpočtovej logiky a výpočtových jadier, ktoré boli taktované na nižších frekvenciách (za posledných 10 rokov sú frekvencie procesorov ustálené na hodnotách v rozmedzí 2-4 GHz).

2.1.1 Taxonómia paralelných počítačov

Už v roku 1966 Michael Flynn klasifikoval paralelné počítače podľa počtu inštrukcií, ktoré je počítač schopný vykonávať súbežne a podľa počtu dátových položiek, ktoré je počítač schopný spracovať konkurentne.

- *Single Instruction, Single Data (SISD)* - Jednoduchý sekvenčný počítač schopný vykonávať jednu inštrukciu a spracovávať jednu dátovú položku.
- *Single Instruction, Multiple Data (SIMD)* - Vektorové procesory schopné spracovať jednou inštrukciou vektor dátových položiek. Na princípe tohto dizajnu pracujú do určitej úrovne súčasné *Graphical Processing Units (GPUs)* a modul SIMD.
- *Multiple Instructions, Single Data (MISD)* - Tento princíp sa využíva v aplikáciach, ktoré vyžadujú vysokú odolnosť voči chybám, kedy dáta sú spracovávané viacerými výpočtovými jednotkami a výsledok je vybraný na základe majoritného princípu.
- *Multiple Instructions, Multiple Data (MIMD)* - Do tejto kategórie spadajú všetky viacjadrové procesory. MIMD procesory sa ďalej zvyknú deliť podľa typu pamäťového priestoru na:
  - *Shared memory MIMD* - Architektúra procesora, ktorá obsahuje globálne prístupný zdieľaný pamäťový priestor, ktorý zjednodušuje všetky transakcie medzi jadrami.

– *Distributed memory MIMD* - Jadrá komunikujú pomocou správ, pričom neexistuje žiadne médium, ktoré by bolo zdieľané medzi viacerými jadrami.

2.1.2 Rýchla vyrovnávacia pamäť (cache)

Pod pojmom cache sa rozumejú malé vysokorýchlostné pamäte (typicky SRAM[1]), ktoré obsahujú úseky hlavnej pamäte, do ktorých sa naposledy pristupovalo. V porovnaní s hlavnou DRAM[2] pamäťou, ktorej doba prístupu je typicky ∼60 ns, sú cache niekoľkonásobne rýchlejšie. Rýchlosť prístupu vyrovnávacích pamätí závisí od typu procesora [15]. Vyrovnávacie pamäte zlepšujú výkonnosť procesorov najmä vďaka konceptu známemu ako referenčná lokalita *(Locality of Reference)*. Referenčná lokalita znamená, že v ľubovoľnom časovom okamihu bude procesor pristupovať k malému alebo obmedzenému úseku hlavnej pamäte. Vyrovnávacia pamäť prednačíta tento úsek a tým umožní rýchlejší prístup procesora do daného úseku.

S rýchlymi vyrovnávacími pamäťami súvisia nasledujúce pojmy:

- *Cache hit* - Nastáva, ak sa požadované dáta nachádzajú v cache.
- *Cache miss* - Opak cache hit. Nastáva, ak sa požadované dáta nenachádzajú v cache.
- *Konzistencia a koherencia cache* - V každom časovom okamihu je nutné zaistiť konzistenciu obsahu cache s obsahom hlavnej pamäte. Konzistenciu zaisťujú metódy *Snooping* (cache kontroluje prenášané adresy a teda dokáže určiť, či sa transakcie snažia o prístup k dátam, ktoré obsahuje) a *Snarfing* (v prípade, že transakcia pracuje s adresami, ktoré zodpovedajú obsahu cache sú dáta prenášané po dátovej zbernici uložené aj do cache)

Z hľadiska efektívneho využívania vyrovnávacích pamätí je nutné si uvedomiť, akým spôsobom sú organizované a ako súvisia s hlavnou pamäťou. Hlavná pamäť je organizovaná rovnomerne do blokov zvaných *cache pages*, ktoré majú veľkosť vyrovnávajúcej pamäte (toto delenie nie je ekvivalentné deleniu na pamäťové stránky). Tieto bloky sú ďalej delené na takzvané *cache lines*, ktorých veľkosť je rôzna v závislosti od typu procesora a dizajnu vyrovnávacej pamäte (typicky 64 bytov). Každý dátový prvok v pamäti má priradenú *cache page* podľa vzorca [1]:

$$cache\ page = \frac{adresa}{veľkosť\ cache\ line}\ \%\ počet\ cache\ pages \tag{2.1}$$

Súčasné procesory obsahujú okrem interných vyrovnávacích pamätí, ktoré sú najbližšie ku každému jadru a typicky aj prístupné len danému jadru (L1 a L2 cache), aj zdieľané vyrovnávacie pamäte (L3 cache).

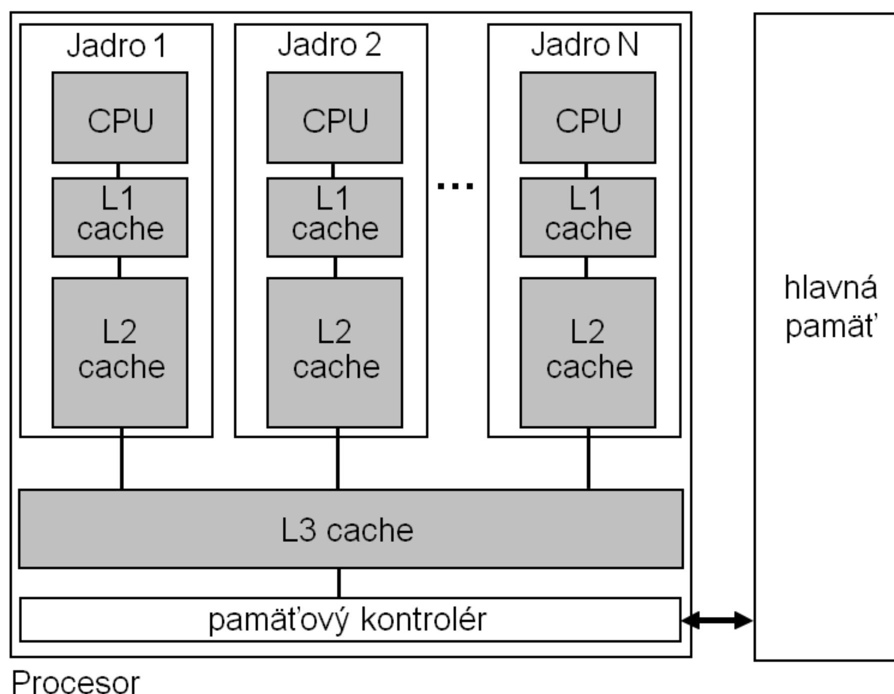
Znalosť systému vyrovnavacích pamätí je nevyhnutná pre efektívnu implementáciu paralelných algoritmov. Najčastejším problémom vznikajúcim v systémoch s distribuovanými vyrovnávacími pamäťami je tzv. *false sharing*. Tento jav nastáva, ak dva alebo viac procesorov operuje nad nezávislými dátami v rovnakom adresovom priestore uložiteľnom v jednej cache line. V tomto prípade môže systém zabezpečujúci koherenciu celej cache po každom zápise donútiť všetky jadrá pristupujúce k jednej cache line k jej synchronizácii napriek tomu, že v rámci cache line je prístup disjunktný. Najúčinnejším riešením tohto problému

[1] static random-access memory [17]
[2] dynamic random-access memory [16]

je zarovnávanie dát v pamäti na veľkosť cache line. To je možné docieliť buď manuálnym odsadzovaním *(padding)* alebo pri dynamickej alokácii použitím funkcií:

```
1 void * _mm_malloc (size_t size, size_t alignment);
2 void   _mm_free(void * p);
```

False sharing je možné eliminovať aj správnym prístupovým vzorom a distribúciou dát.



Obrázok 2.1: Typická pamäťová hierarchia viacjadrového procesora

2.1.3 Vektorové rozšírenia inštrukčnej sady

Súčasné procesory disponujú modulmi pre vektorové spracovanie dát, ktoré je v súčasnosti nevyhnutné napríklad v oblasti počítačovej grafiky (viď 2.1.1). Vektorové spracovanie má obrovské uplatnenie aj v oblasti supercomputingu.

SSE (Streaming SIMD Extensions) [20] Je rozšírenie inštrukčnej sady procesorov Intel o vektorové inštrukcie vhodné pre spracovanie dátovo náročných algoritmov. Predchodcom SSE bola inštrukčná sada MMX[3], ktorá síce poskytovala vektorové inštrukcie, ale zdieľala registrový priestor s FPU[4] a umožňovala prácu len s celými číslami. SSE prinieslo 8 nových 128-bitových registrov v 32-bitovom režime a ďalších 8 registrov v 64-bitovom režime. V pôvodnej verzii SSE umožňovalo vektorové spracovanie 4 32-bitových desatinných čísel. Od verzie SSE2 bola inštrukčná sada rozšírená o možnosť spracovania 2 64-bitových desatinných čísel s dvojitou presnosťou, 4 32-bitových celých čísel, 8 16-bitových celých čísel a 16 8-bitových hodnôt. SSE je podporované aj na procesoroch AMD.

[3] multimedia extensions
[4] floating point unit

AVX (Advanced Vector Extensions) [20] Je rozšírenie inštrukčnej sady procesorov Intel a AMD predstavené v roku 2008. AVX bolo prvýkrát podporované s príchodom architektúry Sandy Bridge v roku 2011. Registrová sada modulu AVX priamo rozširuje registre SSE zo 128-bitovej šírky na 256-bitov a v novej verzii AVX-512 až na 512-bitovú šírku.

2.2 Supercomputing

Pod pojmom supercomputing sa rozumie vykonávanie výpočtovo extrémne náročných algoritmov na superpočítači. Superpočítače na rozdiel od bežných počítačov disponujú obrovskou výpočtovou silou. Procesory sú umiestnené do tzv. uzlov, ktoré sú vzájomne prepojené vysokorýchlostnými spojmi. *Technická univerzita Ostrava* disponuje dvomi superpočítačmi *Salomon* a *Anselm* [8]. Oba superpočítače využívajú na alokáciu zdrojov *PBS Professional*. Popis práce so systémom PBS je v prílohe D.

2.2.1 Salomon

Superpočítač Salomon je zložený z 1008 výpočtových uzlov, z ktorých je 576 bežných a 432 akcelerovaných. Každý uzol obsahuje dva 12-jadrové *Intel Xeon E5-2680v3, 2.5GHz* procesory a je vybavený 128GB RAM. Prístup na superpočítač prebieha prostredníctvom protokolu SSH a pripojením na jeden zo 4 prihlasovacích uzlov.

Procesory *Intel Xeon E5-2680v3* patria pod Intel Haswell architektúru. Každé jadro disponuje 64KB inštrukčnou/dátovou L1 cache, 256KB L2 cache a zdieľanou 30MB L3 cache. Procesor podporuje vektorové rozšírenie AVX2.

Salomon dosahuje teoretickú výkonnosť 2011 TFLOPS (viď 2.5.1).

2.2.2 Anselm

Superpočítač Anselm je zložený z 209 výpočtových uzlov, z ktorých je 180 bežných a 29 akcelerovaných. Každý uzol obsahuje dva 8-jadrové *Intel Sandy Bridge E5-2470, 2.3GHz* procesory a najmenej 64GB RAM. Prístup na Anselm prebieha podobne ako na Salomon.

Procesor *Intel Sandy Bridge E5-2470* obsahuje 256KB L2 cache pre každé jadro a 20MB L3 cache zdieľanú každým jadrom.

Anselm dosahuje teoretickú výkonnosť 94 TFLOPS (viď 2.5.1).

2.3 Optimalizácia kódu

Predchadzajúca podkapitola bola zameraná na prvky moderných procesorov, ako je systém vyrovnávacích pamätí a vektorové rozšírenia. Zmyslom týchto prvkov je poskytnúť softwarovým vývojárom prostriedky, vďaka ktorým môžu vytvárať optimalizované implementácie, ktoré vedú k vyššej výkonnosti. Efektivitu implementácie je možné docieliť rôznymi optimalizačnými technikami ako sú:

1. paralelizácia
2. vektorizácia

3. cache priehľadné algoritmy[5]

Manuálne použitie týchto techník výrazne zvyšuje náročnosť implementácie a zhoršuje čitateľnosť kódu. Programátor musí navyše poznať hardwarové parametre počítača. Aby sa odstránili potenciálne problémy manuálnej implementácie, boli vyvinuté automatické a semi-automatické/asistované optimalizačné prostriedky.

Automatická optimalizácia Zvyčajne prebieha úplne v réžii prekladača. Programátor ovplyvňuje len úroveň optimalizácie a použitie rozšírení pomocou prepínačov pri preklade. Automatická vektorizácia a paralelizácia je v zložitejších aplikáciach len veľmi zriedka použiteľná. Tento typ optimalizácie neovplyvňuje charakter algoritmu a má len minimálny dopad na efektivitu využitia cache. Navýšenie výkonnosti automatickou optimalizáciou nie je zanedbateľné, ale ani dostačujúce. Automatická optimalizácia je vhodná na rýchlu optimalizáciu jednoduchých užívateľských aplikácií.

Asistovaná optimalizácia Spočíva v definovaní spôsobu optimalizácie častí kódu programátorom. Za týmto účelom boli definované viaceré štandardy, medzi ktoré sa radí aj OpenMP. Tieto štandardy poskytujú jednoduché, ale veľmi efektívne rozhranie umožňujúce paralelizáciu a vektorizáciu kódu na podstatne vyššej úrovni s vyššou úrovňou kontroly.

2.4 OpenMP 4.0

OpenMP *(open multi-processing)* [18] je aplikačné programové rozhranie (API), ktoré sa používa primárne na riadenie multithreadového (viac-vláknového) paralelizmu so zdieľanou pamäťou. Jednotlivé vlákna navzájom komunikujú pomocou zdieľaných premenných. Nechcené zdieľanie dát vedie k tzv. *race condition*, ktorý spôsobuje rôzny výstup paralelného programu po každom spustení. Riešením tohto problému je synchronizácia prístupu do zdieľaného dátového priestoru. OpenMP je zložené z troch hlavných komponentov:

- funkcie spúšťané za behu aplikácie
- premenné prostredia
- direktívy prekladača

OpenMP kladie dôraz na:

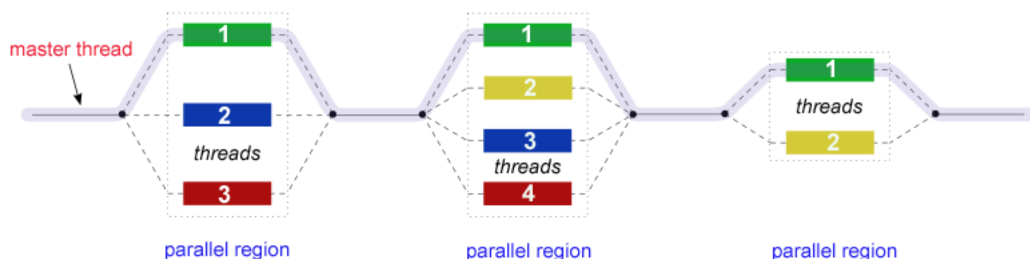
- *Štandardizáciu* - Poskytnutie funkčného štandardu na všetkých typoch architektúr a platforiem. OpenMP je definované a vyvíjané skupinou hlavných predajcov HW a SW.
- *Jednoduchosť použitia* - Vysoká úroveň a kvalita paralelizmu s čo najmenšími nárokmi na programátora.
- *Portabilitu* - API je špecifikované pre C/C++ a Fortran. OpenMP je funkčné na väčšine bežne používaných platformách vrátane Unix/Linux a Windows.

Verzia OpenMP 4.0 bola vydaná v júli 2013 a okrem automatickej paralelizácie poskytuje direktívy aj na asistovanú vektorizáciu.

[5] Z anglického *cache-oblivious/transcendent/friendly* = algoritmy dizajnované za účelom maximálneho využitia cache

2.4.1 Programovací model

Paralelizácia v OpenMP funguje na princípe *fork-join paralelizmu*. To znamená, že sekvenčne spracovávaný program je pomocou OpenMP rozdelený na určitý počet vlákien, z ktorých je každému pridelené identifikačné číslo. Vlákno s ID 0 sa nazýva *master thread* a jedná sa o pôvodné vlákno. Takto vytvorená skupina vlákien sa nazýva *team vlákien*, ktorý pracuje paralelne a na konci paralelnej sekcie sa opäť spojí do jedného vlákna. V rámci paralelnej sekcie je možné vytvárať ďalšie vnorené paralelné sekcie.



Obrázok 2.2: Fork-join paralelizmus [3]

Štandard OpenMP poskytuje jediný možný spôsob vytvorenia paralelnej sekcie a to pomocou direktívy `#pragma omp parallel`. Dáta, ktoré boli alokované mimo paralelnej sekcie sú zdieľané všetkými vláknami a fyzicky sú uložené na hromade v pamäti. Dáta, ktoré boli staticky alokované vo vnútri paralelnej sekcie sú súkromné pre každé vlákno a fyzicky sú uložené na zásobníku.

2.4.2 Runtime funkcie

OpenMP knižničné funkcie spúšťané za behu programu sa delia do troch kategórií:

1. funkcie nastavujúce parametre OpenMP prostredia
2. časovacie funkcie
3. funkcie pre prácu so zámkami

Kompletný zoznam a popis všetkých OpenMP funkcií vrátane nových funkcií, ktoré priniesol štandard 4.0, je v prílohách (viď príloha B).

2.4.3 Premenné prostredia

Napriek tomu, že OpenMP umožňuje dynamicky určovať množstvo parametrov paralelného spracovania, v praxi sa častejšie využíva prostredie a jeho premenné. Premenné prostredia umožňujú efektívne meniť parametre OpenMP prostredia bez nutnosti opakovaného prekladu aplikácie. Všetky premenné prostredia sú písané veľkými písmenami, pričom ich hodnoty nerozlišujú veľké a malé písmená a taktiež ignorujú počiatočné/koncové biele znaky.

Kompletný zoznam a popis všetkých premenných prostredia vrátane nových premenných, ktoré priniesol štandard 4.0, je v prílohách (viď príloha C).

2.4.4 Synchronizácia

OpenMP poskytuje 3 základné high-level synchronizačné konštrukcie:

1. `#pragma omp barrier` - vytvorenie bariéry vo vnútri paralelnej sekcie, ktorá zabraňuje vláknam pokračovať kým sa na bariéru nedostanú všetky vlákna.

2. `#pragma omp critical` - vytvorenie kritickej sekcie, v ktorej sa môže v ľubovoľnom časovom okamihu vyskytovať len jediné vlákno.

3. `#pragma omp atomic` - niektoré často používané jednoduché operácie disponujú hardwarovou podporou. Atomic sa dá chápať ako vytvorenie malej kritickej sekcie, ktorá využije túto hardwarovú podporu, pričom existuje predpoklad, že operácia vykonávaná v tejto konštrukcii je natoľko rýchla, že nie je nutné vytvoriť tak komplexné synchronizačné prostriedky ako pre tradičnú kritickú sekciu. V prípade, že operácia nemá hardwarovú podporu, atomic sa chová rovnako ako critical.

2.4.5 Deľba práce

Deľba práce *(worksharing)* slúži na distribúciu vykonávaného algoritmu medzi jednotlivé vlákna. OpenMP poskytuje možnosť cyklickej deľby práce, rozdelenie práce do sekcií a pridelenie práce práve jednému vláknu.

1. `#pragma omp for` - Pokyn prekladaču rozdeliť nasledujúci cyklus medzi vlákna v paralelnej sekcii. Aby sa dala dosiahnuť vyššia úroveň optimalizácie, je možné k tejto direktíve pridať dodatočný parameter `schedule` (rozvrh), ktorý definuje, akým spôsobom má prekladač rozdeliť iterácie cyklu medzi jednotlivé vlákna.

   - `schedule(static[,chunk])` - Rozdelenie iterácií do blokov o veľkosti chunk a následné rozdelenie týchto blokov medzi jednotlivé vlákna. V prípade, že chunk nie je definovaný, rozdelenie je vykonané rovnomerne na základe počtu vlákien.
   - `schedule(dynamic [,chunk])` - Uloženie iterácií do logického frontu. Každé vlákno vyberie iteráciu zo začiatku frontu a vykoná ju. Hlavný rozdiel oproti static je, že výber iterácií prebieha za behu aplikácie, nie počas prekladu. Veľkosť chunk definuje počet iterácií, ktoré vlákno odoberá zo začiatku logického frontu.
   - `schedule(guided [,chunk])` - Podobné ako dynamic, ale s tým rozdielom, že veľkosť chunk sa počas výpočtu dynamicky znižuje.
   - `schedule(runtime)` - Výber rozvrhu prebieha počas behu programu na základe premennej prostredia `OMP_SCHEDULE` alebo runtime funkcie.
   - `schedule(schedule(auto))` - Automatický výber rozvrhu prekladačom.

2. `#pragma omp sections` - Používa sa na manuálne rozdelenie štrukturovaného bloku kódu na sekcie pomocou `#pragma omp section` blokov. Každý z takto definovaných blokov je vykonaný ľubovoľným vláknom v paralelnej sekcii.

3. `#pragma omp single` - Určuje sekciu kódu, ktorá je vykonaná len jedným, ľubovoľným vláknom. Podobnou konštrukciou je `#pragma omp master`, kedy sekciu kódu vykoná len *master thread*.

Na konci každej spomínanej konštrukcie s výnimkou `#pragma omp master` sa nachádza implicitná bariéra. Programátor má možnosť túto bariéru zrušiť pomocou parametru `nowait`.

2.4.6 Dátový rámec

Napriek tomu, že dáta alokované mimo paralelnej sekcie sú zdieľané a dáta alokované v rámci paralelnej sekcie sú vlastné pre každé vlákno, v niektorých prípadoch je nutné tento atribút dát pozmeniť v záujme správneho fungovania programu. Na tento účel slúžia dátové klauzuly OpenMP definujúce dátový rámec *(data scope)* v rámci paralelnej sekcie alebo v konštrukcii deľby práce. Medzi tieto klauzuly patria:

1. `default(private/shared/none) -` Určuje rámec všetkých premenných, ktoré sa nenachádzajú v žiadnej inej klauzule. `default(none)` je výhodný pri debugovaní programu.
2. `shared(variable list) -` Premenné v zozname sú zdieľané medzi vláknami.
3. `private(variable list) -` Každá premenná v zozname má vytvorenú kópiu v každom vlákne. Táto kópia nie je inicializovaná.
4. `firstprivate(variable list) -` Tento variant rovnako ako `private` vytvorí súkromnú kópiu premennej a inicializuje ju na hodnotu známej premennej nachádzajúcej sa mimo paralelnej sekcie.
5. `lastprivate(variable list) -` Na konci paralelnej sekcie posledné vlákno, ktoré dokončilo svoju činnosť, zmení hodnotu známej premennej nachádzajúcej sa mimo paralelnej sekcie na hodnotu svojej súkromnej kópie predtým, než je táto hodnota vymazaná zo zásobníku.
6. `reduction(operator:variable list) -` Redukcia predstavuje veľmi špecifickú definíciu dátového rámca premennej. Pre všetky premenné, ktoré sa nachádzajú v zozname klauzuly sú vytvorené súkromné kópie na zásobníku každého vlákna. Tieto premenné sú inicializované na základe operátora v klauzule redukcie (0 pre operátor +, 1 pre operátor * ...). Každé vlákno vykonáva program so svojou lokálnou kópiou a po ukončení paralelného bloku, pre ktorý je klauzula definovaná, sú tieto hodnoty agregované do jednej výslednej hodnoty pomocou redukčného operátora, pričom táto hodnota je následne uložená do známej premennej mimo paralelného bloku. Typické použitie redukcie je pri paralelnom výpočte sumy prvkov v poli. Štandard OpenMP 4.0 umožnuje definovať vlastné redukcie.

Redukčný operátor (počiatočná hodnota)			
+	(0)	|	(0)
*	(1)	^	(0)
−	(0)	&&	(1)
&	(~0)	||	(0)
max (Najmenšia hodnota dátového typu redukčného operandu)			
min (Najväčšia hodnota dátového typu redukčného operandu)			

Tabuľka 2.1: Možnosti redukcie v OpenMP 4.0

2.4.7 OpenMP úlohy

Pod pojmom *OpenMP úloha* sa myslí nezávislá jednotka práce, ktorá sa skladá z kódu, dátového prostredia a interných kontrolných premenných, ktoré riadia správanie OpenMP

programu. Spúšťanie týchto úloh je riadené systémom, pričom platí, že všetky úlohy budú dokončené na najbližšej bariére. Úlohy sa v OpenMP vytvárajú pomocou direktívy `#pragma omp task`. Bariéru pre úlohy je možné vytvoriť pomocou `#pragma omp taskwait`. Úlohy podporujú klauzuly dátových rámcov. OpenMP úlohy sú vhodné na riešenie problémov, ktoré nepracujú s poľami dát a na rekurzívne riešené problémy.

2.4.8 Asistovaná vektorizácia

Väčšina moderných prekladačov je schopná automaticky vektorizovať kód pomocou inštrukcií SSE a AVX (viď 2.1.3), no v určitých prípadoch prekladač nie je dostatočne inteligentný. Z tohto dôvodu bola vo verzii OpenMP 4.0 vytvorená podpora asistovanej vektorizácie pomocou dvoch jednoduchých direktív prekladača využívajúcich podobné klauzuly ako predošlé konštrukcie.

1. `#pragma omp simd [14]` - Aplikovaná na cyklus s pevným počtom opakovaní. Počítadlo cyklu musí byť celé číslo. V rámci cyklu sa nesmú nachádzať žiadne ďalšie OpenMP konštrukcie. Program sa nesmie vetviť do vnútra ani von z cyklu. Programátor použitím tejto klauzuly explicitne definuje základné parametre cyklu, ktoré musia byť splnené pre efektívnu vektorizáciu cyklu:

   - invariantnosť premenných v podmienke ukončenia cyklu
   - polia dát podieľajúce sa na výpočte sú v pamäti disjunktné
   - nezávislosť na poradí matematických operácií
   - vektorový výpočet je rýchlejší než skalárny

   Klauzuly:

   - `private(variable list), lastprivate(variable list), reduction(operator:variable list)` - Dátový rámec premenných (viď 2.4.6)
   - `linear(variable1:step1, variable2:step2 ...)` - Premenné v tejto klauzule sú súkromné pre každú iteráciu vektorového spracovania a oznamujú prekladaču, že premenná *variable* je lineárne závislá na hodnote *step*. Hodnota *step* musí ostať počas celého trvania výpočtu nemenná a jej implicitná hodnota je 1.
   - `safelen(length)` - Definuje maximálnu vzdialenosť v logickom priestore medzi prvkami dvoch konkurentných iterácií vektorového spracovania.
   - `aligned(variable1:align1, variable2:align2 ...)` - Oznamuje prekladaču, že objekty na ktoré ukazujú hodnoty premenných *variable* sú v pamäti zarovnané na hodnotu *align*.
   - `collapsed(number)` - Určuje počet vnorených cyklov, ktoré môžu byť spojené do jedného cyklu s väčším počtom iterácií.

2. `#pragma omp declare simd [14]` - Nachádza sa pred definíciou funkcie. Takto definované funkcie je možné volať zvnútra vektorizovaného cyklu. V tomto prípade sú všetky argumenty funkcie spracovávané ako vektory a to isté platí aj o návratovej hodnote. Toto implicitné chovanie je programovo ovplyvniteľné použitím nasledujúcich klauzúl:

- `uniform(variable list)` - Označuje zoznam skalárnych premenných, ktoré sú zdieľané medzi všetkými iteráciami.
- `linear(param1:step1, param2:step2)` - Každý z parametrov v zozname je inkrementovaný o hodnotu *step* - lineárna závislosť.
- `simdlen(number)` - Definuje maximálnu dĺžku vektora, ktorý môže prekladač použiť pri spracovaní argumentov a výsledku funkcie.
- `aligned(argument:align, ...)` - Všetky argumenty v zozname klauzuly sú zarovnané na hodnotu *align*.

2.5 Analýza výkonnosti

Získavanie vyššej výkonnosti je hlavnou motiváciou paralelného a vektorového prístupu. Výkonnosť je charakterizovaná ako vykonané množstvo užitočnej práce v pomere k času a použitými zdrojmi systému. Výkonnosť je kvantifikovaná pomocou rôznych metrík.

2.5.1 Metriky

1. *Zrýchlenie* - Definuje pomer trvania výpočtu sekvenčného programu a jeho paralelizovanej verzie. Čas výpočtu je však ovplyvnený mnohými faktormi ako kvalita implementovaného algoritmu, použitý prekladač, operačný systém či záťaž systému v dobe testovania. Preto je vhodné pri používaní tejto metriky splniť dva základné princípy:

   - Porovnávané programy sú testované na identických softwarových aj hardwarových platformách a za rovnakých podmienok.
   - Sekvenčná verzia programu predstavuje najrýchlejšie známe riešenie problému.

   Výpočet zrýchlenia prebieha podľa vzorca:

$$speedup = \frac{t_{seq}}{t_{par}} \tag{2.2}$$

   Efektivita paralelizovaného riešenia je vyjadrená pomocou vzorca:

$$efficiency = \frac{speedup}{cores\ count} \tag{2.3}$$

2. *IPS (instructions per second)* - Vyjadrenie výkonnosti pomocou počtu inštrukcií za sekundu. Táto metrika je dnes prevažne nepoužívaná z dôvodu jej nepresnosti. IPS neberie do úvahy výkonnosť pamäťovej hierarchie systému, ktorá má výrazný vplyv na celkovú výkonnosť. IPS sa častejšie vyskytuje vo forme MIPS (mega/milión IPS) alebo GIPS (giga/miliarda IPS).

3. *FLOPS (floating point operations per second)* - Je v preklade počet operácií v pohyblivej rádovej čiarke za sekundu. Táto metrika je vhodná na meranie výkonnosti v oblasti vedeckých výpočtov, ktoré využívajú veľké množstvo operácií s reálnymi číslami. FLOPS sa bežne používa na meranie teoretickej výkonnosti procesorov pomocou vzorca:

$$total\ FLOPS = sockets * \frac{cores}{socket} * frequency * \frac{FLOPS}{cycle} \tag{2.4}$$

4. *Priepustnosť pamäte* - Vyjadruje rýchlosť, s ktorou môžu byť dáta čítané alebo zapisované do pamäte procesorom. Najčastejšie sa meria v hodnotách GB/s. Priepustnosť pamäte je základnou metrikou ukazujúcou kvalitu využitia pamäte testovanou implementáciou.

5. *Cache miss/hit rate* - Definuje pomer počtu výpadkov v rýchlej vyrovnávacej pamäti procesora (*cache miss*) a počtu všetkých prístupov do cache (*cache miss*+*cache hits*; viď 2.1.2). Táto metrika odráža kvalitu využitia pamäťovej hierarchie testovanej implementácie.

$$miss\ rate = \frac{misses}{total\ accesses} \tag{2.5}$$

$$hit\ rate = 1 - miss\ rate \tag{2.6}$$

2.5.2 Nástroje

1. *Performance Application Programming Interface (PAPI) [11]* - PAPI implementuje prenosné a efektívne API umožňujúce prístup k počítadlám výkonu hardwaru *(hardware performance counters)*, ktoré je možné nájsť na všetkých moderných procesoroch. PAPI umožňuje monitorovať približne 100 rôznych udalostí (počet podporovaných je rozdielny od architektúry procesora). Procesory však disponujú obmedzeným počtom registrov určených na meranie výkonnosti (∼10), preto PAPI poskytuje možnosť mapovania udalostí na hardwarové počítadlá. PAPI obsahuje high-level rozhranie pre jednoduché ovládanie počítadiel, ktoré sa skladá len z 8 základných funkcií. Za účelom kvalitnejšej analýzy výkonu je možné použiť PAPI low-level rozhranie, ktoré poskytuje vyššiu úroveň kontroly.

   PAPI obsahuje podporu pre viacvláknové aplikácie, ale jeho použitie s OpenMP sa ukázalo ako viac než problematické. Problém vyplýva z toho, že OpenMP vytvára vlákna len raz, typicky na začiatku programu, a v paralelných regiónoch ich len znovu využíva namiesto toho, aby pre každý paralelný región vlákna vytváralo a zatváralo. PAPI monitoruje udalosti pre každé vlákno (nie pre každé jadro) a preto je podstatné, aby boli funkcie nastavujúce PAPI a dátové štruktúry využívané PAPI volané/vytvárané v správnom poradí vzhľadom na vytváranie a využívanie vlákien.

   Najbezpečnejším spôsobom využívania PAPI v spojitosti s OpenMP je vytvorenie globálnych premenných určených pre uloženie `EventSetov`[6] a hodnôt počítadiel na začiatku programu. Tieto premenné je nutné označiť ako `#pragma omp threadprivate`, aby boli vytvorené súkromne pre každé vlákno v okamihu jeho vytvorenia. Inicializácia PAPI a jeho podpory vlákien musí prebehnúť pred prvou paralelnou sekciou. Napriek všetkým opatreniam výsledky nie sú vždy v OpenMP aplikáciach spoľahlivé.

2. *Allinea MAP [7]* - Profiler pre paralelné, viacvláknové alebo sekvenčné C, C++, Fortran a F90 programy. Allinea je špeciálne vyvíjaná, aby bola schopná profilovať pthreads, OpenMP a MPI v paralelizovaných programoch. Na rozdiel od PAPI nie je nutné vkladať analýzu výkonnosti do kódu. Allinea pracuje priamo so spustiteľným binárnym súborom preloženým s prepínačom `-g`. Vďaka jednoduchosti používania

[6] Premenná typu `int` obsahujúca identifikátor množiny PAPI udalostí

a kvality výsledkov je Allinea jedným z najlepších nástrojov na meranie a analýzu výkonnosti.

3. *Intel VTune Amplifier [10]* - VTune je dizajnovaný na pohodlné a efektívne meranie parametrov implementácie na Intel procesoroch. Užívateľ si môže prostredníctvom príjemného grafického rozhrania zobraziť analýzu výkonnosti jednovláknových aj paralelných aplikácií, ktorá obsahuje výpočet priepustnosti pamäte, analýzu zámkov a energetických nárokov. Najnovšie verzie dokážu odhadnúť miesta v programe, kde je možné získať vyšší výkon. Intel VTune Amplifier plne podporuje štandard OpenMP.

Kapitola 3

Praktická časť

Väčšina problémov je riešiteľná nespočetným množstvom rôznych algoritmov, ktoré sa vyznačujú rôznymi vlastnosťami. Bezhlavá paralelizácia kódu často vedie k poklesu výkonnosti a strate správnosti výsledkov. Cieľom tejto kapitoly nie je len ukázať výborné optimalizačné vlastnosti štandardu OpenMP, ale aj poukázať na dôležitosť jeho správneho využívania. Všetky benchmarky boli preložené pomocou Intel C++ prekladača. Prehľad dôležitých prepínačov z hľadiska optimalizácie je možné nájsť v prílohe E. Všetky grafy boli vytvorené pomocou programu Gnuplot [9]. Vo všetkých benchmarkoch sa pracuje s dátovým typom `float`[1]. Výnimkou je binárne vyhľadávanie (viď 3.4), v ktorom sú hodnoty uzlov dátového typu `long long int`[2].

3.1 Automatická a asistovaná vektorizácia

Prekladače poskytujú celú škálu možností automatickej vektorizácie kódu, ktorá je spustená pomocou prepínačov povoľujúcich rôzne úrovne optimalizácie. Prekladače navyše poskytujú možnosť zobrazenia výsledku optimalizácie prekladu počas rôznych fáz vrátane fázy vektorizácie. Pomocou týchto nástrojov je možné vyhodnotiť dopad automatickej vektorizácie a identifikovať časti kódu, ktoré vyžadujú asistovanú vektorizáciu.

Vlastnosti vektorizácie je možné sledovať na jednoduchej funkcii:

```
1 void ex(float * A, float * B, float * C,
2         float * D, float * E, int N) {
3     for(int i = 0; i < N; ++i)
4       A[i] = A[i] + B[i] + C[i] + D[i] + E[i];
5 }
```

Preklad:

```
icpc -O3 -qopt-report -qopt-report-phase=vec -opt-report-file=stdout
```

Výstup prekladača:

```
remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below.
Use level 5 report for details
```

[1] desatinné číslo s jednoduchou presnosťou uložené na 4 bytoch
[2] 8-bytové celé číslo

```
remark #15346: vector dependence: assumed FLOW dependence
between line 28 and line 28
```

Z výstupu je zrejmé, že cyklus vo vnútri funkcie nebol vektorizovaný, z dôvodu možných dátových závislostí. Toto je spôsobené tým, že prekladač pri vyššom počte prístupov do vektorov v rámci každej iterácie nevykonáva kontrolu závislostí medzi jednotlivými poliami. V príklade sa v každej iterácii pristupuje až do 5 polí. Programátor má možnosť oznámiť prekladaču, že dané polia nie sú vzájomne závislé. Jedným riešením je využitie kľúčového slova `restrict`. Tento variant vyžaduje explicitné určenie všetkých disjunktných polí v hlavičke funkcie a preklad pomocou prepínača `-restrict`.

```
1 void ex_res(float * restrict A, float * restrict B,
2             float * restrict C, float * restrict D,
3             float * restrict E, int N);
```

Výstup prekladača:

```
remark #15300: LOOP WAS VECTORIZED
```

Toto riešenie však neposkytuje vysokú úroveň flexibility, pretože sa zaoberá len jedným faktorom vplývajúcim na automatickú vektorizáciu. Elegantnejším riešením je využitie asistovanej vektorizácie poskytovanej štandardom OpenMP 4.0. Vektorizácia v tomto prípade nebude mať žiadny vplyv na deklaráciu funkcie, ale bude priamo referovať na cyklus, ktorý je vhodný na vektorizáciu a to pomocou `#pragma omp simd` (viď 2.4.8).

Výstup prekladača:

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
```

Je v tomto prípade asistovaná vektorizácia efektívnejšia než automatická? Odpoveď je nie, pretože v závere sa kód preloží do rovnakých vektorových inštrukcií. Systém však pri využívaní automatickej vektorizácie vykonáva kontrolu dátových závislostí počas behu. Použitím asistovanej vektorizácie sa toto chovanie odstráni, čo môže mať za následok nárast výkonnosti oproti automaticky vektorizovanej verzii.

Metóda	Doba behu [ms]
Nevektorizovaná	1,860
Restrict	0,973
OpenMP 4.0	0,972

Tabuľka 3.1: Priemerná doba behu jedného volania funkcie nad vstupnými vektormi o veľkosti 1 000 000 prvkov

Vektorizácia poskytla zrýchlenie približne o 100%. Rýchlosť vektorizácie závisí najmä od typu inštrukcií, do ktorých bol kód preložený. Preto je vhodné zaistiť, aby prekladač využil najrýchlejší vhodný modul vykonávajúci vektorizáciu (AVX). Prekladače toto umožňujú prepínačom `-mavx` alebo `-xavx`. Pokiaľ ani jeden z prepínačov nie je definovaný, Intel compiler implicitne použije `-msse2`.

Metóda	Doba behu [ms]
Nevektorizovaná	1,846
Restrict	0,906
OpenMP 4.0	0,905

Tabuľka 3.2: Priemerná doba behu jedného volania funkcie nad vstupnými vektormi o veľkosti 1 000 000 prvkov s povoleným AVX. Táto verzia vykazuje nárast efektivity o takmer 10%

3.2 Jednoduché operácie nad vektormi

Väčšina optimalizačných problémov je založená na iteratívnom spracovávaní vektorov a polí, ktoré sa vyznačujú súvislým uložením v pamäti. Tieto operácie majú zvyčajne matematický charakter, vďaka čomu ich je možné nie len paralelizovať, ale aj vektorizovať. Takéto spracovanie je typické najmä v počítačovej grafike, simuláciách a vedeckých aplikáciách pracujúcich s veľkým objemom dát [19]. OpenMP umožňuje veľmi jednoducho rozdeliť jednotlivé iterácie medzi dostupné jadrá. Vlastnosti optimalizácie pomocou OpenMP sú otestované na jednoduchom násobení vektora konštantou a na skalárnom súčine dvoch vektorov.

3.2.1 Násobenie konštantou

V tomto benchmarku sa vykonáva jednoduché násobenie vektora konštantou podľa vzorca:

$$\forall \imath : A_\imath = c * B_\imath \tag{3.1}$$

Jedná sa o jednoduchý počítaný cyklus. V jazyku C má algoritmus podobu:

```
1 for(int i = 0; i < N; ++i)
2   A[i] = c * B[i];
```

Takýto kód je možné paralelizovať aj vektorizovať pridaním jedinej direktívy. Nakoľko sa jedná o direktívu prekladača, kód bude preložiteľný aj na strojoch, ktoré nepodporujú OpenMP. V takom prípade je direktíva ignorovaná a výsledkom je plne sekvenčný kód. Tento cyklus je možné vektorizovať pridaním `#pragma omp simd aligned(A:64, B:64)` a paralelizovať pridaním `#pragma omp parallel for default(shared) schedule(static)`.

Výsledný plne optimalizovaný kód má podobu:

```
1 #pragma omp parallel for simd default(shared) schedule(static) firstprivate(c)
2 for(int i = 0; i < N; ++i)
3   A[i] = c * B[i];
```

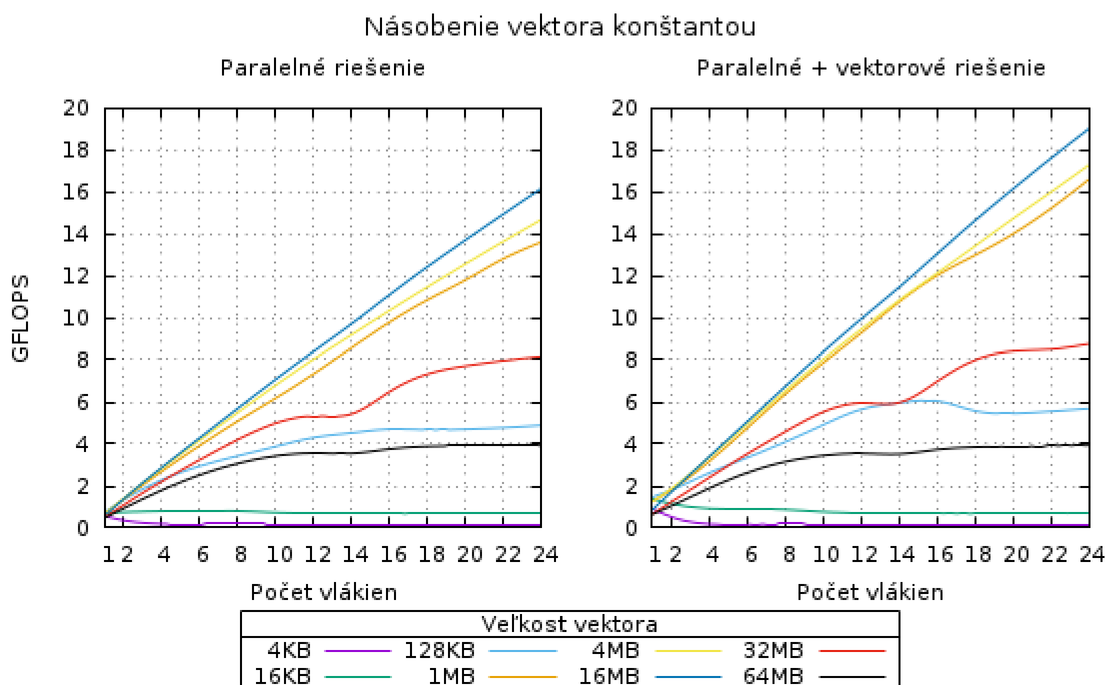
Pred spustením je nutné program preložiť s potrebnými prepínačmi. Neoptimalizovaná verzia bola preložená pomocou *(a)*, optimalizované verzie pomocou *(b)*.

```
(a)  icpc -O3 --restrict -no-vec
(b)  icpc -O3 --restrict -qopenmp -xavx
```

Operácia bola testovaná pre sekvenčné, vektorizované, paralelizované a plne optimalizované riešenie a to na vektoroch o veľkosti 4KB až 64MB. Výsledky získané spracovaním na Salomone (viď 2.2.1):

Veľkosť vektora		4KB	16KB	128KB	1MB	4MB	16MB	32MB	64MB
MFLOPS	SK	780,8	761,1	699,3	564,7	565,2	493,6	451,9	460,4
	VK	2564,9	2793,8	1754,1	1365	1354,1	844,3	667,2	674,7
	PK	167,6	708,4	4899	13550,6	14669,6	16155,8	8160	3908,6
	PVK	157,3	724	5689,3	16620,3	17286,8	19003,9	8807,3	3937,8

Tabuľka 3.3: Nameraná výkonnosť sekvenčného (SK) a vektorizovaného kódu (VK) v MFLOPS na jednom jadre, paralelného (PK) a paralelného + vektorizovaného kódu (PVK) na 24 jadrách. Tabuľka ukazuje veľmi mierne klesajúcu výkonnosť sekvenčnej verzie bez vektorizácie v závislosti na veľkosti vstupného vektora. Vektorizácia zvýšila výkonnosť pre menšie rozmery vektorov až 4-násobne. Tento nárast výkonnosti sa však so zväčšujúcim vstupným vektorom znižuje. Dôvodom znižovania výkonnosti je problematickejšie využívanie pamäťovej hierarchie. Tabuľka jasne ukazuje vplyv paralelizmu na výkonnosť v závislosti od veľkosti vektora.



Obrázok 3.1: Porovnanie výkonnosti rôznych úrovní optimalizácie násobenia vektora konštantov. Ľavý graf ukazuje výkonnosť paralelizovanej verzie algoritmu. Pravý graf ukazuje výkonnosť paralelizovanej a vektorizovanej verzie algoritmu. Na grafoch je vidieť očakávaný absolútny pokles výkonnosti pre menšie vstupné vektory. Paralelizácia nadobúda efekt až pre vektory väčšie než 32KB (veľkosť L1 cache). Od veľkosti približne 16MB sa stráca výkonnosť kvôli pamäti. Na grafoch je znateľný vplyv vektorizácie, ktorá v najlepšom prípade navyšuje výkonnosť až o 3 GFLOPS. Na grafoch je taktiež vidieť, že pre veľmi veľké vektory (64MB+) sa výrazne stráca vplyv paralelizmu už na 10 jadrách. To je spôsobené dosiahnutím maximálnej priepustnosti pamäte pre danú operáciu. Zvyšných 14 jadier je možné využiť na vykonávanie inej činnosti, alebo ich odpojiť, aby sa znížila spotreba energie.

3.2.2 Skalárny súčin

Skalárny súčin vektorov prebieha podľa vzorca:

$$sum = \sum_{i=0}^{N-1} (A_i * B_i) \tag{3.2}$$

Opäť sa jedná o jednoduchý počítaný cyklus. V jazyku C má algoritmus podobu:

```
float sum = 0.0f;
for(int i = 0; i < N; ++i)
  sum += A[i] * B[i];
```

Na rozdiel od predchádzajúceho benchmarku nie sú všetky iterácie úplne nezávislé. Každá iterácia pristupuje ku zdieľanej premennej `sum`, pričom do nej priamo zapisuje. Preto je potrebné vytvorenie privátnej premennej pre každé vlákno, do ktorej budú zaznamenávané medzivýsledky skalárneho súčinu v rámci každého vlákna. Tieto výsledky budú následne agregované do výslednej hodnoty výsledku. Túto činnosť je možné zaistiť pomocou redukcie (2.4.6). Vektorizácia prebieha pridaním `#pragma omp simd reduction(+:sum) aligned(A:64, B:64)` a paralelizácia pridaním `#pragma omp parallel for reduction(+:sum) default(shared) schedule(static)`.

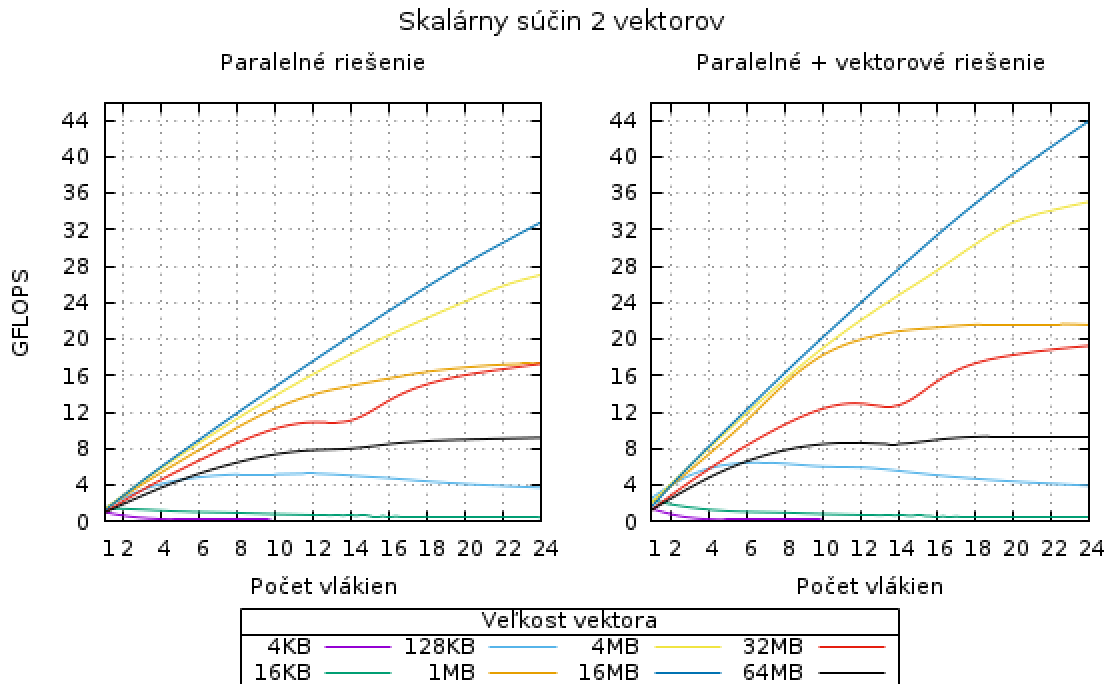
Výsledný plne optimalizovaný kód má podobu:

```
float sum = 0.0f;
#pragma omp parallel for simd reduction(+:sum) default(shared) schedule(static)
for(int i = 0; i < N; ++i)
  sum += A[i] * B[i];
```

Preklad prebieha rovnako ako v prvom benchmarku. Operácia bola testovaná pre sekvenčné, vektorizované, paralelizované a plne optimalizované riešenie a to na vektoroch o veľkosti 4KB až 64MB. Výsledky získané spracovaním na Salomone:

	Veľkosť vektorov	4KB	16KB	128KB	1MB	4MB	16MB	32MB	64MB
MFLOPS	SK	2147,6	2207,6	1933,4	1573,2	1573,8	1407,5	1323,7	1353,6
	VK	3670,2	5251,4	5872,2	4824,6	4753,5	2966	2437,7	2551,7
	PK	122	473,2	3751,3	17310,6	27001,3	32866,7	17337,2	9232,7
	PVK	121,9	477,6	3985,7	21563,8	35052,4	43918,3	19284,5	9396,4

Tabuľka 3.4: Nameraná výkonnosť sekvenčného (SK) a vektorizovaného kódu (VK) v MFLOPS na jednom jadre, paralelného (PK) a paralelného + vektorizovaného kódu (PVK) na 24 jadrách. Tabuľka opäť ukazuje mierny pokles výkonnosti so vzrastajúcou veľkosťou vektorov. Vektorizácia sa najviac preukazuje pri veľkostiach vstupných vektorov od 16KB do 4MB a potom klesá na približne dvojnásobok sekvenčnej verzie. Dôvodom poklesu výkonnosti je problematickejšie využívanie pamäťovej hierarchie



Obrázok 3.2: Porovnanie výkonnosti rôznych úrovní optimalizácie skalárneho súčinu 2 vektorov. Ľavý graf ukazuje výkonnosť paralelizovanej verzie algoritmu. Pravý graf ukazuje výkonnosť paralelizovanej a vektorizovanej verzie algoritmu. Grafy ukazuju rovnaké vlastnosti optimalizácie ako grafy na obrázku 3.2.1. Skalárny súčin však dosahuje dvojnásobnú výkonnosť. To je spôsobuje využívanie operácií násobenia aj sčítania (nie len násobenia ako je to u násobenia konštantou), čo umožňuje vykonávať dvojnásobný počet operácií za rovnaký čas. Na grafoch je opäť pozorovateľné ustálenie výkonnosti pre veľké vektory (64MB+) už na 10 jadrách. Toto ustálenie je spôsobené dosiahnutím maximálnej pamäťovej priepustnosti pre danú operáciu.

3.3 Mergesort

Vyhľadávanie a radenie patria medzi typické operácie návrhu algoritmov. Medzi najznámejšie formy radenia patrí aj metóda zvaná *mergesort*. V najbežnejšej podobe sa jedná o rekurzívny radiaci algoritmus, ktorý v prvej fáze počas procesu rekurzívneho zanorenia delí vstupné pole na menšie časti, kým nedosiahne maximálne rozdelenie na polia jednotkovej dĺžky. Tie sú potom počas procesu rekurzívneho vynorenia spájané tak, aby vzniklo zoradené pole.

Z pohľadu optimalizácie je jasnou komplikáciou neiteratívne spracovanie vstupného poľa. Za účelom paralelizácie rekurzívnych algoritmov a algoritmov pracujúcich nad nesúvislými dátovými štruktúrami boli v OpenMP vytvorené úlohy (viď 2.4.7). Implementácia sekvenčnej verzie mergesortu sa skladá z funkcie `merge`, ktorá na princípe *insertsortu* spája dve zoradené polia a z funkcie `merge_sort`, ktorá sa v svojom tele rekúrzivne volá nad polovicami vstupného poľa a následne tieto polovice radí pomocou funkcie `merge`.

Použitý algoritmus pracuje okrem pôvodného vstupného poľa aj s jeho kópiou. Funkcia `merge` vždy zlučuje prvky z jedného vstupného poĺa do druhého, čím sa odstraňuje potreba

neefektívneho preusporiadania prvkov v rámci jedného poľa alebo vytvárania dočasných pomocných polí na ukladanie medzivýsledkov. Týmto spôsobom sa zvyšuje pamäťová náročnosť, ale aj výkonnosť.

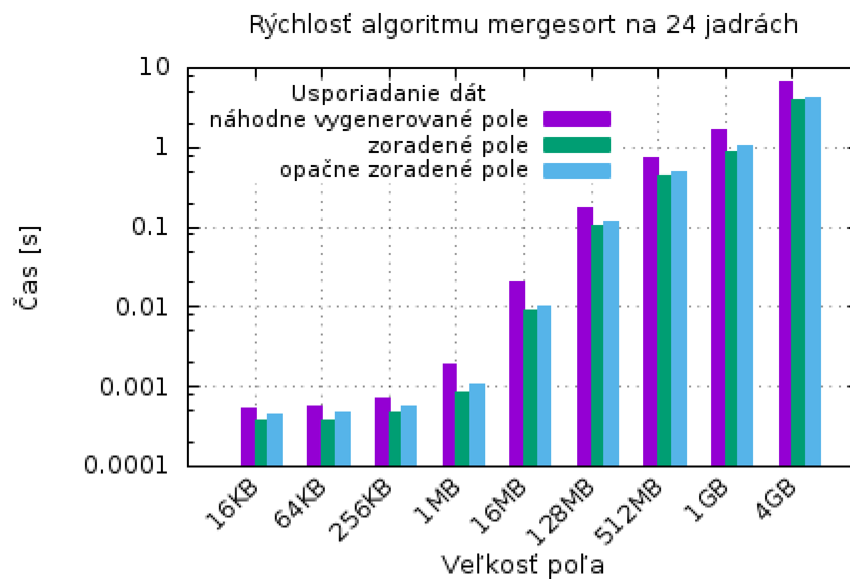
Paralelizácia pomocou OpenMP prebieha pridaním direktív `#pragma omp task` a `#pragma omp taskwait` do sekvenčnej implementácie nasledovným spôsobom:

```
int merge_sort(float * A1, float * A2, int N) {
  if(N < MIN_TASK_SIZE) {
    std::sort(&A1[0], &A1[N]);
    return RESULT_IN_A1;
  }

  #pragma omp task shared(left_status)
  int left_status = merge_sort(&A1[0], &A2[0], N/2);

  #pragma omp task shared(right_status)
  int right_status = merge_sort(&A1[N/2], &A2[N/2], N - N/2);

  #pragma omp taskwait

  /*
   Zlúčenie polovíc do jedného z polí A1 a A2
   na základe výsledku volaní funkcií merge_sort
  */
  ... }
```

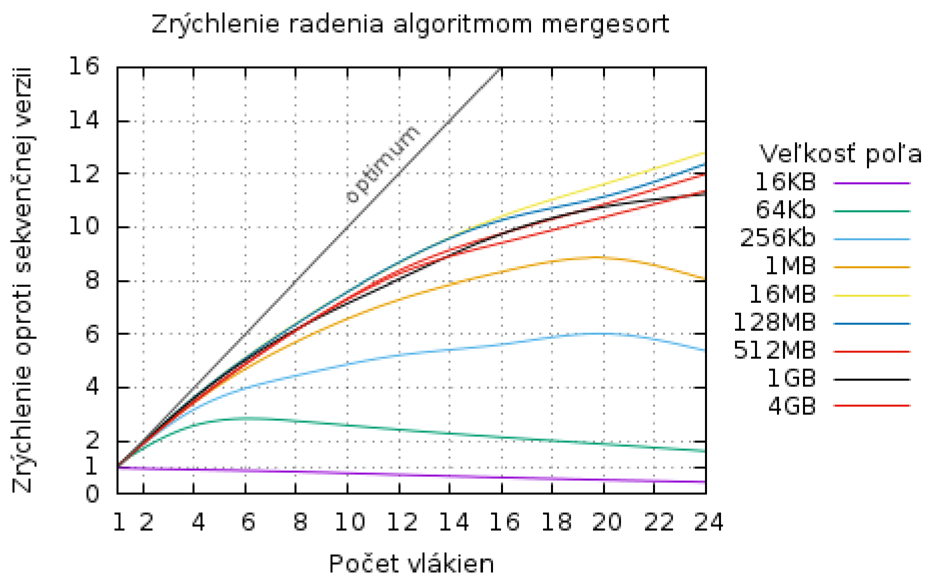
Aby paralelizácia nadobudla efekt, musí byť funkcia `merge_sort` volaná v tele direktívy `#pragma omp single`. Počas rekurzívneho zanorovania sú postupne vytvárané OpenMP úlohy. Aby sa zabránilo nadmernému počtu príliš malých úloh, ktoré by spôsobili úpadok výkonnosti, je na začiatku tela funkcie vykonávaná kontrola veľkosti vstupného poľa. Ak je vstupné pole menšie než je veľkosť L1 cache ($\simeq$ 8000 4-bytových položiek), tak je vykonané sekvenčné radenie pomocou funkcie `std::sort`. Direktíva `taskwait` zaisťuje vykonanie všetkých predošlých úloh a teda zaručuje zlúčenie len zoradených častí poľa.

Na výkonnosť radiacich algoritmov má vo všeobecnosti značný vplyv prediktor skokov[3] [5]. To je spôsobené vysokým počtom operácií porovnania hodnôt pri radení. Spôsob usporiadania prvkov v poli má preto značný vplyv na rýchlosť zoradenia daného poľa. Pokiaľ sú podmienené príkazy vyhodnocované v rozpoznateľnom vzore, prediktor skokov je schopný tento vzor nájsť a efektívnejšie predpovedať výsledok podmienky. Preto sa radiace algoritmy testujú na rôzne inicializovaných poliach.

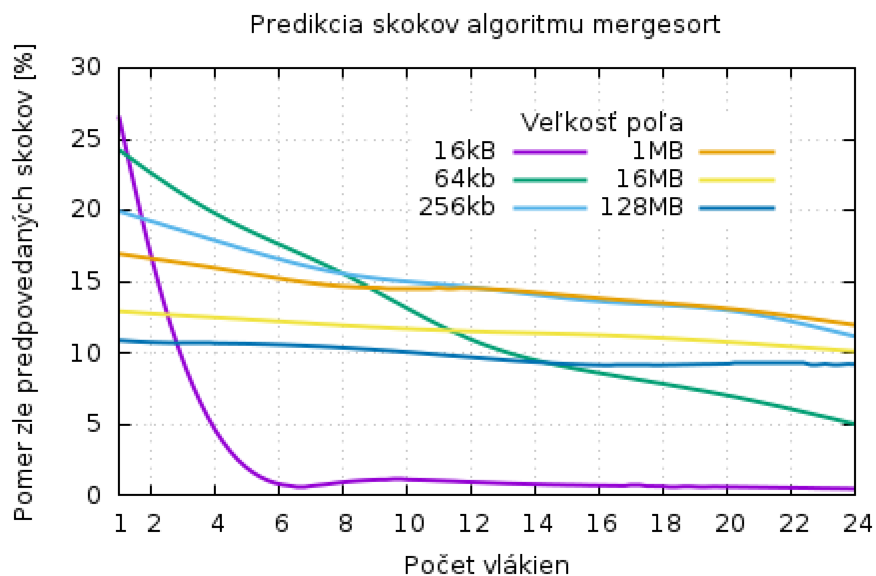
[3]digitálny obvod, ktorý sa snaží dopredu predpovedať výsledok podmieneného príkazu `if-else`



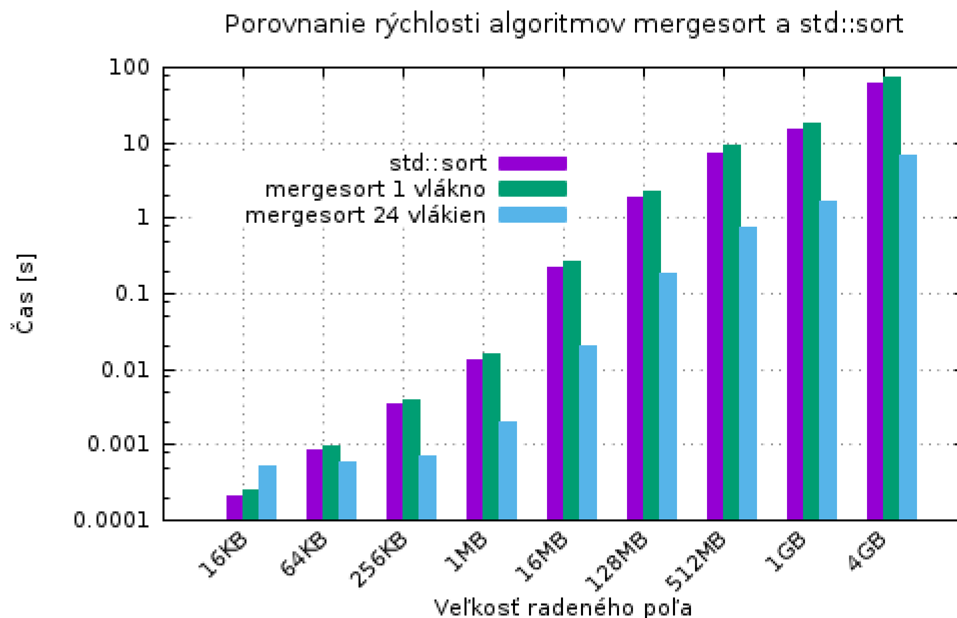
Obrázok 3.3: Histogram rýchlosti algoritmu mergesort v závislosti od veľkosti radeného poľa na 24 jadrách. Na histograme je viditeľná dlhšia doba spracovania náhodne vygenerovaného poľa a to bez ohľadu na veľkosť poľa až takmer o $\frac{1}{4}$ rádu. Zvislá časová os má logaritmickú mierku so základom 10. Najrýchlejšie je podľa očakávania radenie už zoradeného poľa. Z činnosti prediktoru skokov vyplýva aj vysoká rýchlosť radenia opačne zoradeného poľa.



Obrázok 3.4: Graf zrýchlenia algoritmu mergesort pomocou OpenMP úloh. Na grafe je vidieť postupný nárast zrýchlenia s narastajúcou veľkosťou vstupného poľa. Malé vstupné polia sú spracované prevažne sekvenčne na jednom jadre. Prvky vstupného poľa boli generované náhodne.



Obrázok 3.5: Graf predikcie skokov skokov algoritmu mergesort pre náhodne zoradené pole. Na grafe je viditeľné zníženie podielu zle predpovedaných skokov s rastúcim počtom vlákien. Vo všeobecnosti sa podiel zle predpovedaných skokov pohybuje v rozmedzí $10 - 20\%$. Pre zoradené a opačne zoradené polia je podiel zle predpovedaných skokov na úrovni $< 1\%$, z čoho vyplýva aj nižšia doba výpočtu.

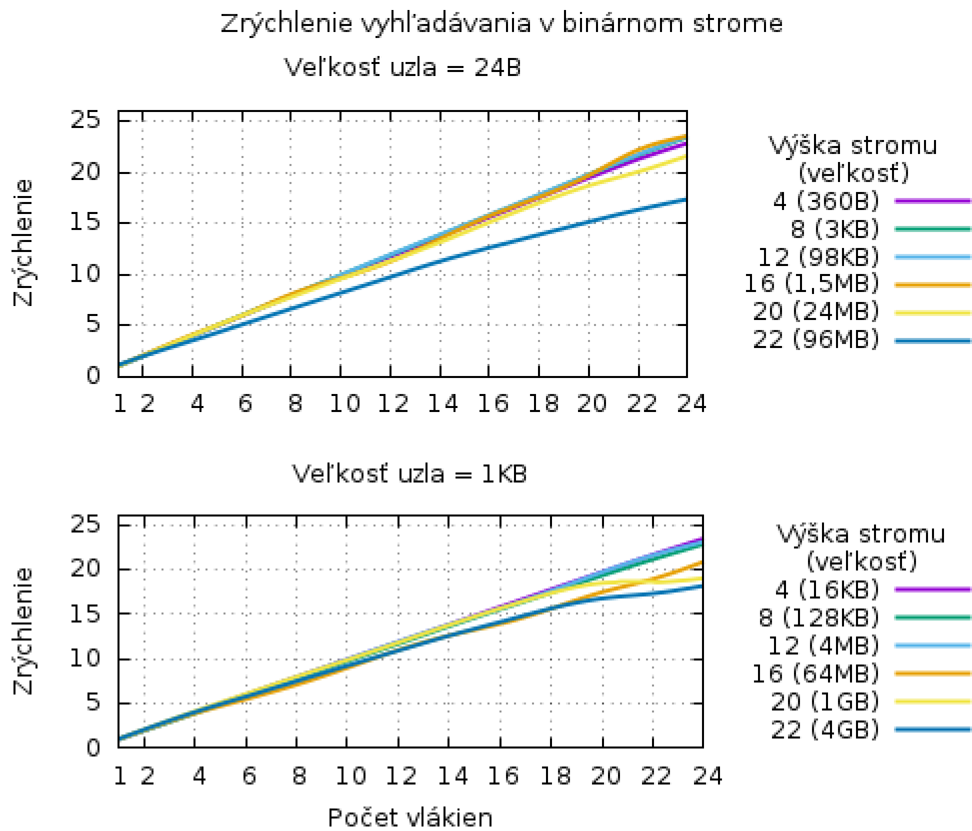


Obrázok 3.6: Porovnanie implementovaného mergesortu a knižničnej funkcie `std::sort`. Zvislá časová os má logaritmickú mierku so základom 10.

3.4 Binárne vyhľadávanie

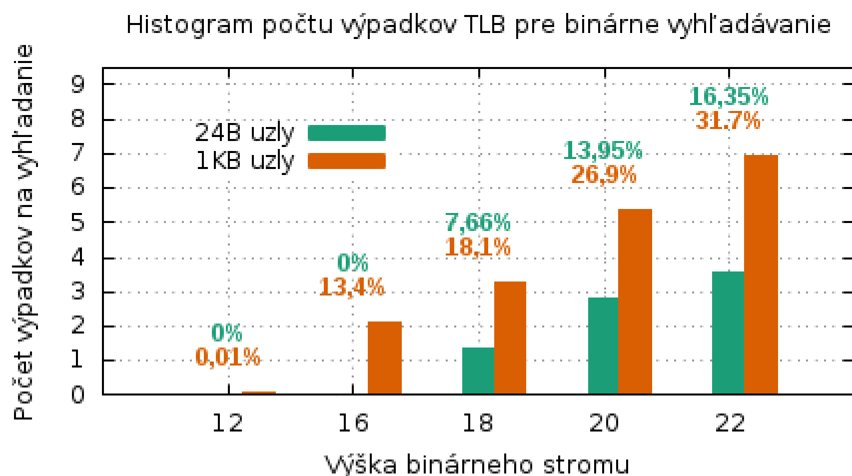
Tento benchmark spočíva v paralelnom vyhľadávaní veľkého počtu položiek v jednom vyváženom binárnom strome. Paralelizácia prebieha jednoduchým `#pragma omp for` cyklom, v ktorom sa postupne prechádza pole 10^6 položiek typu `long long int`, ktoré sú vyhľadávané v binárnom strome. Vygenerovaný binárny strom je výškovo vyvážený, pričom každý uzol je dynamicky alokovaná štruktúra veľkosti najmenej 24 bytov (8-bytový kľúč a dva smerníky na ľavý a pravý podstrom).

Počas procesu vyhľadania ľubovoľnej položky binárneho stromu o veľkosti N bytov sa vykoná maximálne $\log_2 N$ porovnaní, pričom po každom neúspešom porovnaní je nutné získať adresu ľavého alebo pravého poduzla. Rýchlosť tejto operácie závisí na rozložení prvkov v pamäti. V prípade veľmi veľkých uzlov stromu je pravdepodobné, že takýto strom bude v pamäti uložený s veľkými rozostupmi, čo sa odzrkadlí na počte výpadkov v TLB[4]. Naopak pre malé uzly je pravdepodobnejšie, že TLB bude obsahovať preklady adries oboch podstromov, čím sa zvýši rýchlosť vyhľadania.

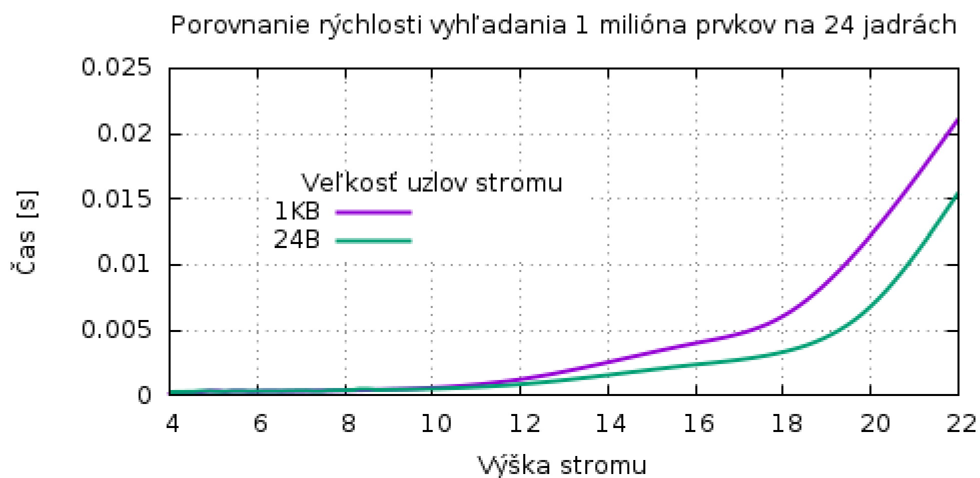


Obrázok 3.7: Grafy zrýchlenia binárneho vyhľadávania. Na grafoch je vidieť postupný pokles výkonnosti paralelizácie so zväčšujúcim sa stromom. Tento pokles výkonnosti spôsobujú najmä výpadky TLB a chybná predpoveď skokov. Grafy ukazujú rovnaké vlastnosti paralelizmu bez ohľadu na veľkosť uzlov.

[4]Translation Lookaside Buffer - cache určená na zrýchlenie prekladu virtuálnych adries



Obrázok 3.8: Histogram priemerného počtu výpadkov v TLB na jedno vyhľadanie v binárnom strome o výške 12 až 22 úrovní. Pre menšie stromy je priemerný počet výpadkov TLB na jedno vyhľadanie zanedbateľný. Nad stĺpcami histogramu sa nachádza percentuálne vyjadrenie pomeru výpadkov TLB ku všetkým prístupom do TLB. Histogram potvrdzuje tvrdenie, že počet výpadkov narastá s veľkosťou uzlov. Počet použitých jadier nemá na hodnoty priemerného počtu výpadkov vplyv.



Obrázok 3.9: Graf priemernej doby vyhľadania 1 milióna prvkov v binárnom strome v závislosti na jeho výške. Vplyv výpadkov TLB sa preukazuje aj na dobe výpočtu, ktorá je kratšia pre malé uzly.

3.5 Maticový súčin

Maticový súčin $A \times B = C$ je operácia nad dvomi maticami, pričom každý prvok výslednej matice je výsledkom skalárneho súčinu príslušného riadka matice A a stĺpca matice B. Maticový súčin je teda operácia so zložitosťou $\mathcal{O}(n^3)$, tzn. s kubickou zložitosťou. Táto operácia predstavuje jeden z najzákladnejších optimalizačných problémov, najmä kvôli problematickému, nesúvislému prístupu do pamäte a teda aj neefektívnemu využitiu rýchlych vyrovnávacích pamätí. Na maticovom súčine je možné otestovať rôzne optimalizačné techniky, ako sú transpozícia či blokovanie.

$$A_{\imath,\jmath} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,\jmath} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,\jmath} \\ \vdots & \vdots & \ddots & \vdots \\ a_{\imath,1} & a_{\imath,2} & \cdots & a_{\imath,\jmath} \end{bmatrix} \qquad B_{\jmath,\ell} = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,\ell} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,\ell} \\ \vdots & \vdots & \ddots & \vdots \\ b_{\jmath,1} & b_{\jmath,2} & \cdots & b_{\jmath,\ell} \end{bmatrix} \tag{3.3}$$

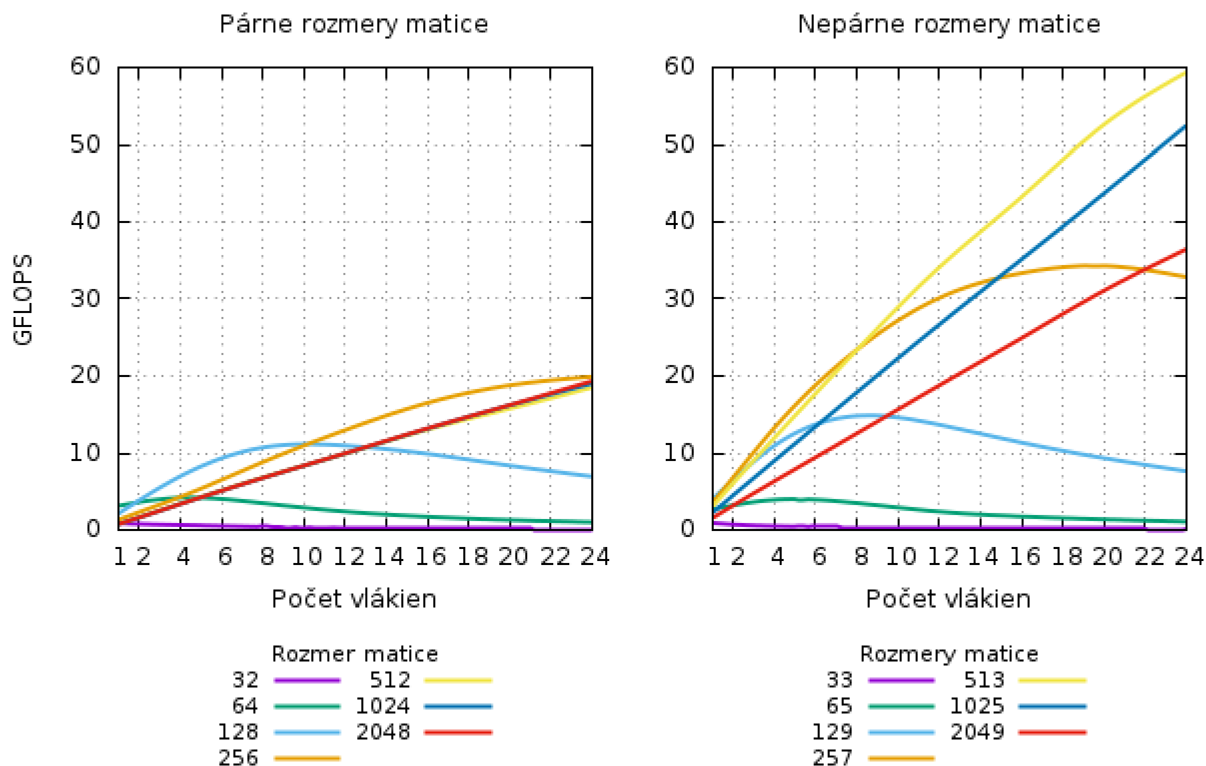
$$C_{\imath,\ell} = A \times B = \begin{bmatrix} \sum\limits_{x=1}^{\jmath}(a_{1,x} * b_{x,1}) & \sum\limits_{x=1}^{\jmath}(a_{1,x} * b_{x,2}) & \cdots & \sum\limits_{x=1}^{\jmath}(a_{1,x} * b_{x,\ell}) \\ \sum\limits_{x=1}^{\jmath}(a_{2,x} * b_{x,1}) & \sum\limits_{x=1}^{\jmath}(a_{2,x} * b_{x,2}) & \cdots & \sum\limits_{x=1}^{\jmath}(a_{2,x} * b_{x,\ell}) \\ \vdots & \vdots & \ddots & \vdots \\ \sum\limits_{x=1}^{\jmath}(a_{\imath,x} * b_{x,1}) & \sum\limits_{x=1}^{\jmath}(a_{\imath,x} * b_{x,2}) & \cdots & \sum\limits_{x=1}^{\jmath}(a_{\imath,x} * b_{x,\ell}) \end{bmatrix} \tag{3.4}$$

3.5.1 Naivný algoritmus

Základným algoritmom riešenia tejto operácie je algoritmus IJK, často nazývaný aj ako naivné násobenie matíc. IJK postupne prechádza všetky prvky matice A po riadkoch a prvky matice B po stĺpcoch.

```
1  #pragma omp parallel for schedule(static)
2  for(int i = 0; i < N; ++i)
3  for(int j = 0; j < N; ++j)
4  {
5    float temp = 0.0f;
6    #pragma omp simd reduction(+:temp)
7    for(int k = 0; k < N; ++k)
8      temp += A[i * N + k] * B[k * N + j];
9    C[i * N + j] = temp;
10 }
```

Tento algoritmus je najmä vďaka svojej jednoduchosti a intuitívnosti využívaný najčastejšie. Z toho vyplýva aj jeho pomenovanie *naivný*, pretože sa spomedzi všetkých algoritmov maticového súčinu radí medzi tie najhoršie.

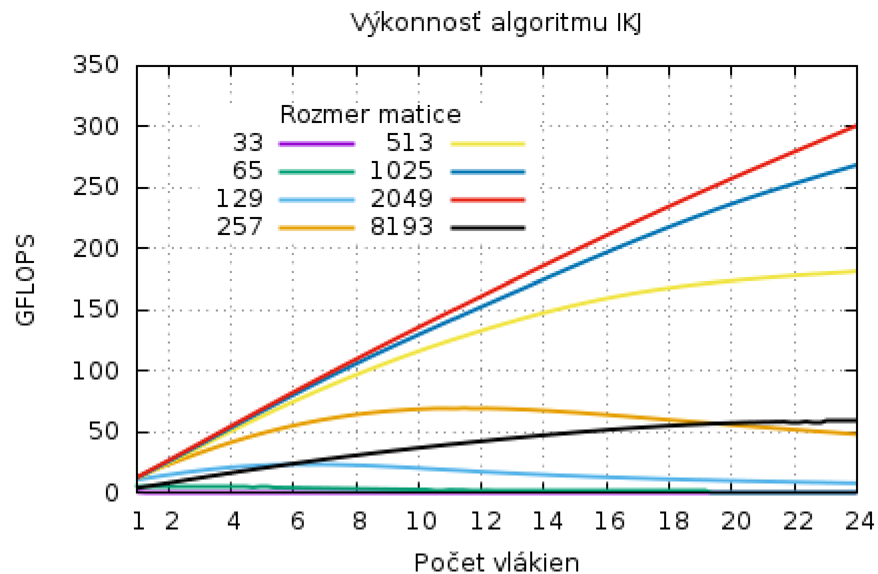


Obrázok 3.10: Grafy výkonnosti algoritmu IJK. Ľavý graf zobrazuje výkonnosť algoritmu IJK pre rozmery, ktoré sú násobkami/mocninami čísla 2. Pravý graf zobrazuje výkonnosť pre tie isté rozmery +1. Porovnanie týchto dvoch grafov odhaľuje najväčší nedostatok algoritmu IJK. Napriek zarovnaniu prvkov na veľkosť *cache line* dochádza k *false sharing* pre matice o rozmeroch mocnín čísla 2. Dochádza k tomu pretože adresa každého prvku matíc je namapovaná na cache stránky podľa vzorca 2.1. Veľkosti *cache line* a počty *cache page* sú v hodnotách mocnín čísla 2, a preto sa môže stať, že prvky z rôznych riadkov matíc budú napriek zarovnaniu namapované na rovnakú *cache page*. Preto je z hľadiska testovania výkonnosti operácií ako je maticový súčin potrebné pracovať s veľkosťami a rozmermi, ktoré tento problém odstraňujú. Pre veľmi malé veľkosti strany matice paralelizácia neprináša žiadne navýšenie výkonnosti, ale ju práve naopak znižuje. Tento jav je viditeľný na maticach do veľkosti $257 \times 257 \simeq 264$KB, pre ktoré je dosiahnutá maximálna výkonnosť už na 18 jadrách. Pre väčšie matice začína byť neefektivita prístupu do matice B viditeľná na postupnom úpadku výkonnosti. Pre násobenie veľkých matíc je teda algoritmus IJK nevhodný.

Neefektivita algoritmu IJK sa odzrkadľuje aj na počte výpadkov cache najmä na úrovniach L1 a L2 (viď. príloha G). IJK dosahuje najvyššiu výkonnosť 60 GFLOPS pre veľkosť matíc $513 \times 513 \simeq 1$MB.

Zlepšenie efektivity je možné docieliť jednoduchou transformáciou IJK algoritmu na IKJ, ktorý poskytuje lepšiu lokalitu prístupu.

```
#pragma omp parallel for schedule(static)
for(int i = 0; i < N; ++i)
for(int k = 0; k < N; ++k)
{
  float temp = A[i * N + k]f;
  #pragma omp simd
  for(j = 0; j < N; ++j)
    C[i * N + j] += temp * B[k * N + j];
}
```



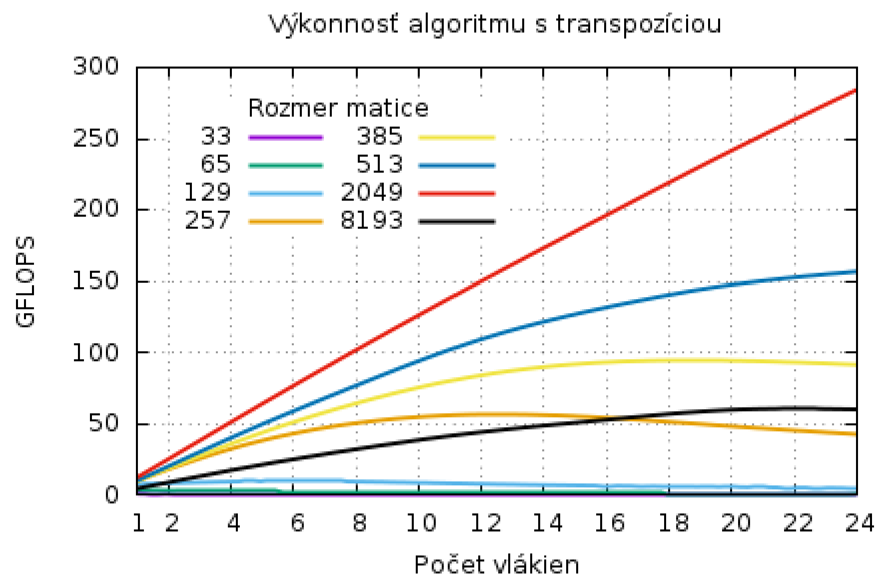
Obrázok 3.11: Graf výkonnosti algoritmu IKJ. Jednoduchá zmena lokality prístupu poskytuje niekoľkonásobný nárast výkonnosti. Maximálna výkonnosť je dosiahnutá pre matice $2049 \times 2049 \simeq 16$MB a to na úrovni 300 GFLOPS. Kvalita lokality prístupu algortimu IKJ sa stráca pre veľmi veľké matice, kedy už pre matice o veľkosti ~ 256MB je výkonnosť len 50 GFLOPS.

3.5.2 Algoritmus s využitím transponovanej matice

Ďalšou možnou modifikáciou algoritmu IJK je využitie transpozície matice B. Transpozícia je prevrátenie matice okolo hlavnej diagonály. Vďaka tomu je možné vynásobiť matice spôsobom, pri ktorom sa do každej z matíc pristupuje len po riadkoch. Takéto riešenie však na úkor lepšej lokality prístupu zvyšuje zložitosť celej operácie o transpozíciu jednej z matíc.

```
// Transpozícia
#pragma omp parallel for
for(int i = 0; i < N; ++i)
for(int j = i + 1; j < N; ++j)
  std::swap(B[i * N + j], B[j * N + i]);
```

```
// Maticový súčin
#pragma omp parallel for schedule(static)
for(int i = 0; i < N; ++i)
for(int j = 0; j < N; ++j)
{
  float temp = 0.0f;
  #pragma omp simd reduction(+:temp)
  for (int k = 0; k < N; ++k)
    temp += A[i * N + k] * B[j * N + k];
  C[i * N + j] = temp;
}
```



Obrázok 3.12: Graf výkonnosti algoritmu využívajúceho transpozíciu matice B. Graf ukazuje podstatné zlepšenie v porovnaní s algoritmom IJK. Algoritmus s transpozíciou dosahuje maximálnu výkonnosť tak isto ako algoritmus IKJ pre matice o veľkosti 2049×2049 na úrovni 280 GFLOPS. Výkonnosť tohto algoritmu od tejto úrovne klesá podobne ako v predošlých algoritmoch z rovnakých dôvodov.

V porovnaní s algoritmom IKJ je algoritmus s transponovanou maticou implementačne zložitejší a vykazuje o trochu horšie výsledky. Výsledok benchmarku však dokazuje vplyv zmysluplného preusporiadania prvkov za účelom lepšej lokality prístupu a teda využitia cache na výkonnosť.

3.5.3 Blokový algoritmus

Najväčším problémom práce s veľkým množstvom dát uloženým v maticiach je, že od určitej veľkosti týchto matíc prestane byť systém vyrovnávacích pamätí schopný efektívne prednačítať dáta z hlavnej pamäte a udržiavať temporálnu lokalitu. To má za následok obrovskú stratu výkonnosti. Veľmi vhodným riešením je tzv. blokovanie. Za predpokladu,

že matice A a B majú formu podľa rovnice 3.3 a teda je ich možné rozdeliť na podmatice do tvaru:

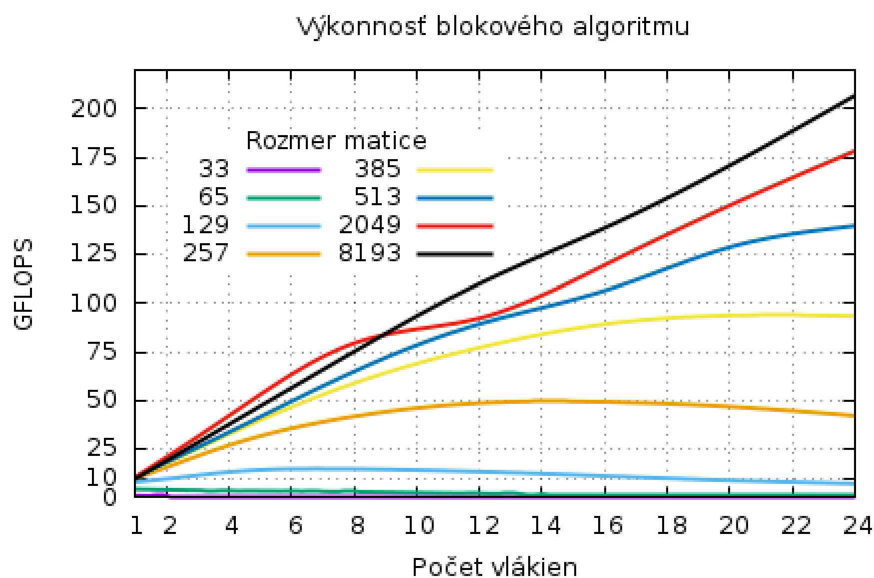
$$A_{\imath,\jmath} = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,j} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ A_{i,1} & A_{i,2} & \cdots & A_{i,j} \end{bmatrix} \qquad B_{\jmath,\ell} = \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ B_{j,1} & B_{j,2} & \cdots & B_{j,l} \end{bmatrix} \tag{3.5}$$

Potom výslednú maticu C je možné vypočítať ako:

$$C_{\imath,\ell} = \begin{bmatrix} \sum_{x=1}^{j}(A_{1,x} \times B_{x,1}) & \sum_{x=1}^{j}(A_{1,x} \times B_{x,2}) & \cdots & \sum_{x=1}^{j}(A_{1,x} \times B_{x,l}) \\ \sum_{x=1}^{j}(A_{2,x} \times B_{x,1}) & \sum_{x=1}^{j}(A_{2,x} \times B_{x,2}) & \cdots & \sum_{x=1}^{j}(A_{2,x} \times B_{x,l}) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{x=1}^{j}(A_{i,x} \times B_{x,1}) & \sum_{x=1}^{j}(A_{i,x} \times B_{x,2}) & \cdots & \sum_{x=1}^{j}(A_{i,x} \times B_{x,l}) \end{bmatrix} \tag{3.6}$$

Táto vlastnosť maticového súčinu umožňuje vykonávať blokovanie. Bežné násobenie matíc potrebuje vykonať $\imath * \ell * \jmath$ súčinov. Každý z týchto súčinov potrebuje prečítať dáta z matíc A a B a zapísať výsledok do matice C. Každý z týchto súčinov je síce možné vykonať paralelne, ale priradenie jednej takejto mikroskopickej operácie pracujúcej nad 3 dátovými prvkami o veľkosti bežných registrov nie je efektívne v okamihu, keď sa všetky dáta nezmestia do nižších úrovní cache. Navyše celkovo $\jmath$ z týchto paralelizovateľných operácií predchádza zápis do jedného prvku výslednej matice, čím sa vytvára potenciál na *race condition* a teda ďalšie zníženie výkonnosti. Týmito problémami trpia všetky predchádzajúce riešenia.

Blokovaním sa počet operácií, ktoré je možné vykonať paralelne, zníži z $\imath * \ell * \jmath$ na $i * l * j$ pričom platí, že $i \leq \imath; l \leq \ell; j \leq \jmath$. Znížením počtu paralelne vykonávateľných operácií sa implicitne navyšuje komplexnosť týchto operácií. Veľkou výhodou však je, že takéto rozdeľovanie práce na menšie celky vytvára možnosť práce s lokálnymi kópiami blokov na každom jadre. Týmto spôsobom si každé jadro prevezme v každej iterácii block matice A aj B a vykoná lokálne maticový súčin, pričom má garanciu toho, že pri zmysluplnej veľkosti blokov budú všetky prvky, ku ktorým pristupuje, uložené v jeho L1 alebo L2 cache. Následne naraz zapisuje väčšie množstvo dát, pričom pri rovnomernom rozložení práce je *race condition* značne eliminovaná. Implementácia tejto verzie algoritmu (viď príloha F) je menej intuitívna, ale výsledná dosiahnutá výkonnosť hovorí sama za seba.



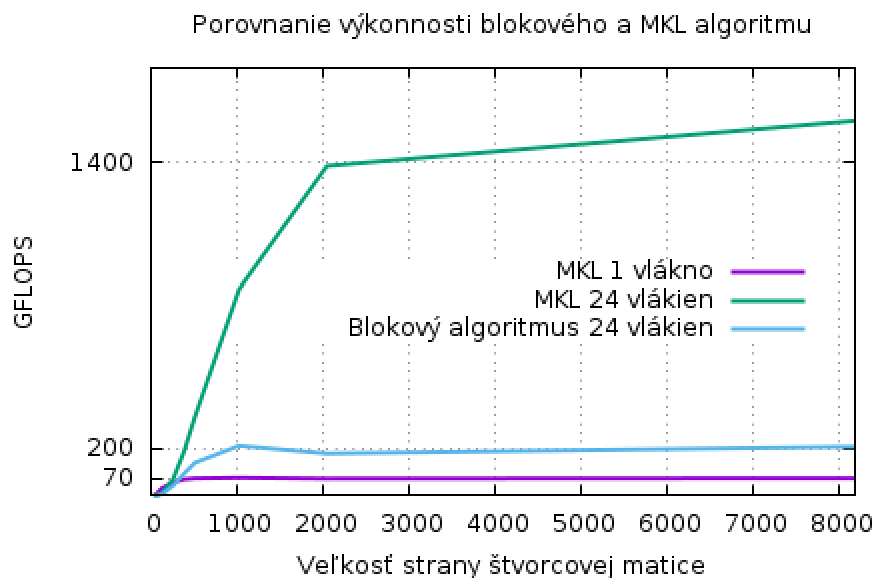
Obrázok 3.13: Graf výkonnosti blokového algoritmu. Tento algoritmus ako jediný ukazuje ustálenie výkonnosti aj pre veľké matice a to na úrovni ∼200 GFLOPS. Pri meraniach bola používaná veľkosť bloku 64×64, čo predstavuje spracovávanie práve $64*64*4*3 \simeq 49$KB dát, ktoré sa prevažne zmestia do 32KB L1 cache. Táto optimalizácia poskytuje čiastočne nižšiu výkonnosť v porovnaní s predošlými algoritmami pre menšie matice. Obrovskou výhodou však ostáva vysoká výkonnosť pri násobení veľkých matíc.

Tento spôsob riešenia maticového výpočtu je z hľadiska využitia OpenMP najlepší. Medzi ďalšie optimalizácie patrí rozvíjanie cyklov *(loop unrolling)* alebo využitie nových algoritmov, ako je Strassenov algoritmus so zložitosťou $\mathcal{O}(n^{2,804})$. Lepšie optimalizácie je potenciálne nutné implementovať až na úrovni assembleru. Vďaka týmto optimalizáciam je možné dosiahnuť až ∼ 8-násobné navýšenie výkonnosti oproti blokovej verzii s využitím OpenMP. Vo všeobecnosti sú takto vysoko optimalizované metódy často implementované len vo veľmi špecializovaných knižniciach ako napríklad *MKL*[5]. Tieto knižnice však nie sú voľne dostupné a ich cena sa pohybuje v stovkách eur.

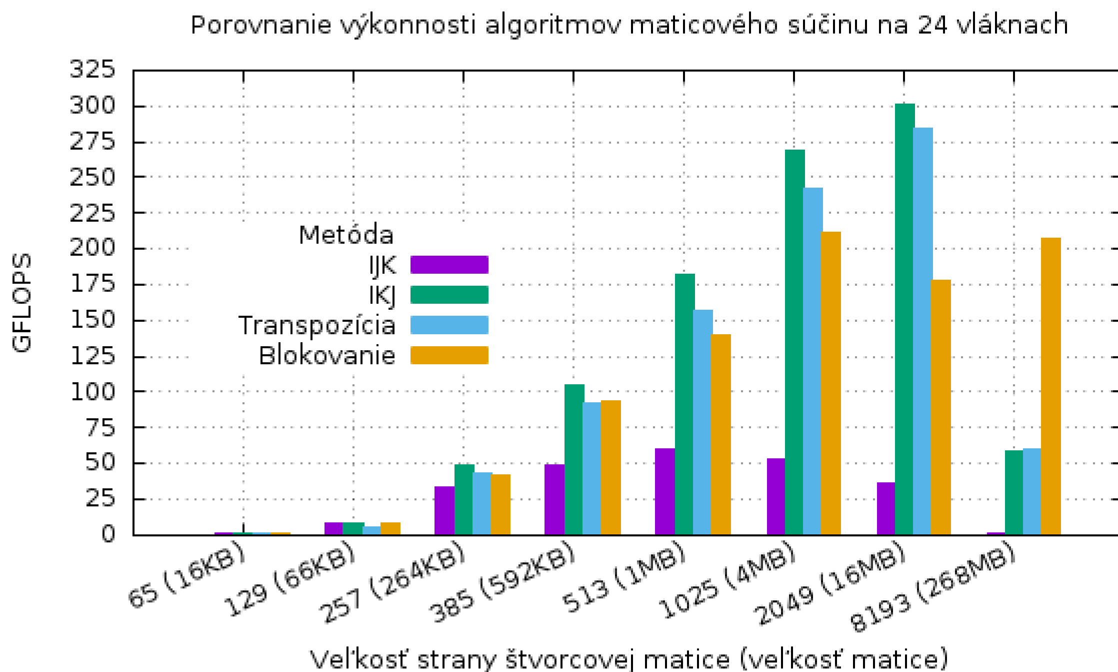
Blokový algoritmus je možné ďalej optimalizovať pre všetky úrovne cache. V rámci výpočtu jednotlivých blokov je možné implementovať všetky spomínané optimalizačné techniky. V prílohe F sa nachádza základná implementácia univerzálnej[6] verzie blokového algoritmu, ktorá bola testovaná v tomto benchmarku.

Porovnanie implementácií násobenia matíc z hľadiska efektivity využívania cache je možné nájsť v prílohe G.

[5]Math Kernel Library
[6]nezávislej od veľkosti matíc ani blokov



Obrázok 3.14: Porovnanie výkonnosti blokového algoritmu a implementácie z knižnice MKL. V priemere dosahuje MKL $\sim$ 8-násobnú výkonnosť. Z hľadiska ručnej implementácie je však použitý blokový algoritmus podstatne jednoduchší a predstavuje vhodnú alternatívu, pretože ako jediný si dokáže vďaka blokovaniu dokáže udržať stabilnú výkonnosť bez ohľadu na veľkosť matice.



Obrázok 3.15: Histogram porovnania rôznych implementácií násobenia matíc.

3.6 Numerické riešenie LaPlaceovej rovnice

Numerické metódy v matematike sú spravidla vhodné na algoritmické riešenie pomocou počítača, pretože pozostávajú z veľkého množstva relatívne jednoduchých operácií. Takéto metódy sú častokrát súčasťou rôznych fyzikálnych, chemických a matematických simulácií, ktoré požadujú vysokú presnosť výpočtu a zároveň aj krátku výpočetnú dobu. Najvhodnejším riešením je implementácia efektívnej optimalizácie paralelizáciou a vektorizáciou. Predmetom tohto benchmarku je numerické riešenie LaPlaceovej rovnice pri výpočte elektrického napätia v štvorcovej oblasti na základe hodnôt elektrického potenciálu na hraniciach tejto oblasti.

U_H

	10	10	10	10	10	10	
7.5							5
7.5							5
7.5							5
7.5							5
7.5							5
7.5							5
	0	0	0	0	0	0	

U_L (vľavo), U_P (vpravo), U_D (dole)

Obrázok 3.16: Problém výpočtu elektrického napätia na ploche, pre ktorú sú hodnoty hraničných potenciálov $U_L = 7,5V$, $U_H = 10V$, $U_P = 5V$ a $U_D = 0V$.

Numerické riešenie spočíva v rozdelení plochy (matice) na čo najmenšie časti (prvky matice) tak, aby sa dalo predpokladať, že zmena potenciálu naprieč každou časťou plochy je zanedbateľná. Následne je možné iterovať cez celú maticu a postupne počítať hodnoty každého prvku na základe hodnôt okolitých prvkov podľa vzorca 3.7 odvodeného v prílohe H, až kým sa nedosiahne požadovaná presnosť výpočtu.

$$U(x, y) = \frac{U(x - \Delta x, y) + U(x + \Delta x, y) + U(x, y - \Delta y) + U(x, y + \Delta y)}{4} \tag{3.7}$$

U_H

	10	10	10	10	10	10	
7.5	8.5	8.7	8.7	8.5	8.2	7.3	5
7.5	7.7	7.6	7.5	7.2	6.8	6.1	5
7.5	7.1	6.7	6.3	6.1	5.8	5.4	5
7.5	6.5	5.7	5.2	4.9	4.8	4.8	5
7.5	5.6	4.4	3.8	3.6	3.6	4.1	5
7.5	3.9	2.6	2.1	1.9	2.1	2.8	5
	0	0	0	0	0	0	

U_L U_P

U_D

Obrázok 3.17: Výsledok príkladu s presnosťou *0.01* po aplikovaní vzorca 3.7.

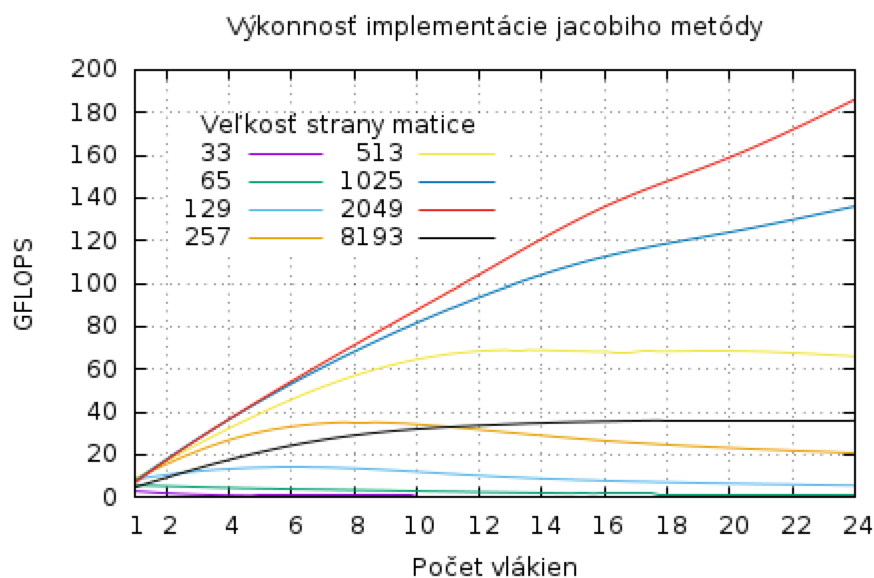
3.6.1 Jacobiho metóda

Jacobiho metóda počíta novú hodnotu každého prvku matice na základne hodnôt z predchádzajúcej iterácie. Tento spôsob vynucuje používanie druhej matice na ukladanie nových hodnôt, pretože prepisovanie hodnôt v pôvodnej matici by ovplyvnilo výpočet v rámci iterácie. Tým, že táto metóda v každej iterácii len číta hodnoty z jednej z matíc, poskytuje ideálny priestor pre využitie efektívnej paralelizácie a vektorizácie.

Výpočet prebieha nasledovne:

```
1  for(i = 0; i < I; ++i) {        // I = maximálny počet iterácií
2    max_E = 0.0f;                 // max_E = maximálna odchýlka v iterácii
3    #pragma omp parallel for default(shared) schedule(static) reduction(max:max_E)
4    for(int row = 1; row < N - 1; ++row)
5    {
6      #pragma omp simd aligned(MM:64, M:64)
7      for(int col = 1; col < N - 1; ++col)
8      {
9        MM[row * N + col] = 0.25f * ( M[(row - 1) * N + col] +
10                                     M[(row + 1) * N + col] +
11                                     M[row * N + col - 1] +
12                                     M[row * N + col + 1] );
13       max_E = MAX[7](max_E, ABS[8](M[row * N + col] - MM[row * N + col]));
14     }
15   }
16   SWAP(M,MM);                   // MM = sekundárna matica
17   if(max_E < E) break;          // E = požadovaná presnosť
18 }
```

[7]funkcia `std::max`, ktorá vyberie maximum spomedzi 2 hodnôt
[8]funkcia `std::fabs`, ktorá vypočíta absolútnu hodnotu parametra funkcie



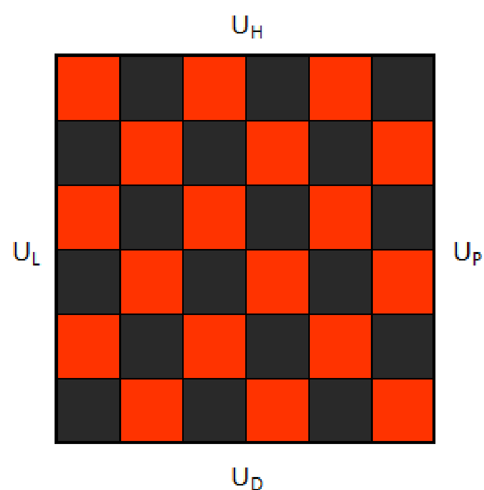
Obrázok 3.18: Graf výkonnosti Jacobiho metódy. Benchmark bol spustený s parametrami $E = 0, I = 20$. Pre menšie hodnoty matíc je úroveň kvality paralelizácie veľmi nízka. Postupne sa však navyšuje pričom najlepšie výsledky pre matice o veľkosti až $2000 \times 2000 \simeq$ 15MB. Následne výkonnosť prudko klesá najmä kvôli neefektívnej práci s pamäťou. Veľký podiel nadobudnutej výkonnosti nesie vektorizácia, ktorá poskytla až 3-násobný nárast výkonnosti oproti čisto paralelnej verzii algoritmu.

Tým, že Jacobiho metóda nevyužíva novovypočítané hodnoty v rámci každej iterácie, síce uľahčuje optimalizáciu, ale podstatne znižuje rýchlosť konvergencie výpočtu k správnemu výsledku.

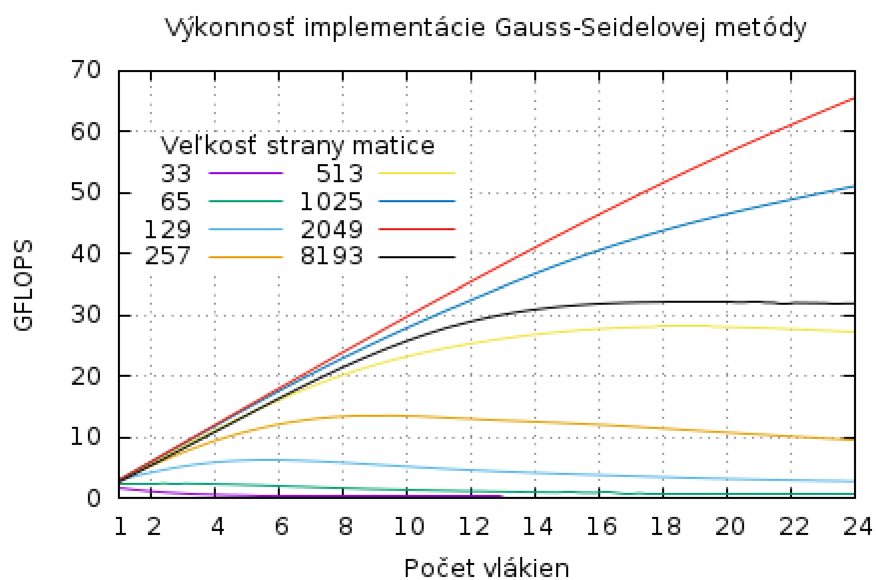
3.6.2 Gauss-Seidelova metóda

Gauss-Seidelova metóda narozdiel od Jacobiho metódy využíva novovypočítané hodnoty v rámci každej iterácie, čím veľmi efektívne zlepšuje úroveň konvergencie celého výpočtu a zároveň odstraňuje podmienku potreby sekundárnej matice, čím sa znižujú pamäťové nároky aplikácie o 50%. Táto zmena však predstavuje obrovský optimalizačný problém, pretože vznikajú sekvenčné závislosti medzi prvkami matice, čím sa odstraňuje možnosť priamej paralelizácie.

Riešením je tzv. červeno-čierne zafarbovanie prvkov matice [4]. Prvky matice sú šachovnicovým spôsobom rozdelené do dvoch skupín. Pre každý *červený* prvok potom platí, že výpočet jeho hodnoty je závislý len na hodnotách *čiernych* prvkov a naopak. Vďaka tejto klasifikácii je možné rozdeliť výpočet nových hodnôt do dvoch cyklov počítajúcich hodnoty prvkov len jednej farby, pričom platí, že obe cykly sú paralelizovateľné.

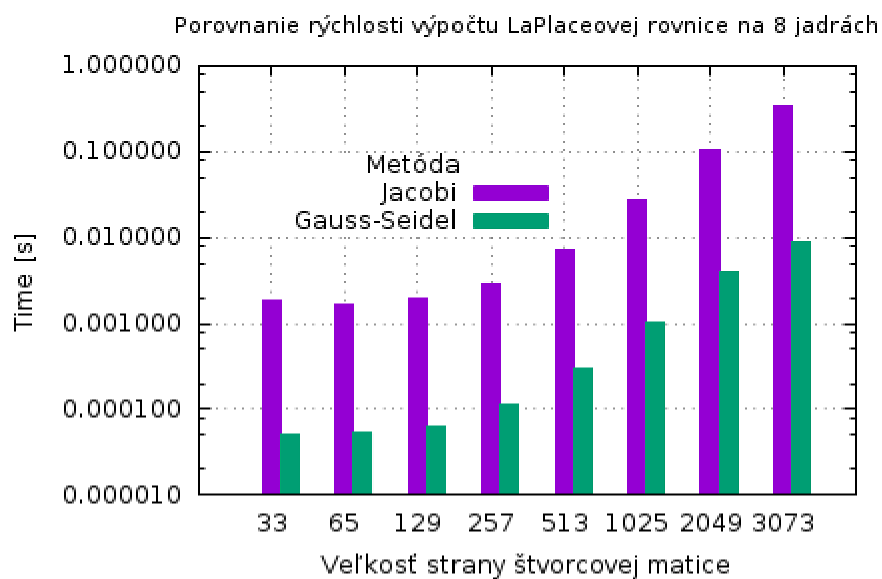


Obrázok 3.19: Ukážka červeno-čierneho farbenia prvkov.



Obrázok 3.20: Graf výkonnosti Gauss-Seidelovej metódy. Benchmark bol spustený s parametrami $E = 0, I = 20$.

Algoritmus dosahuje podstatne nižšiu výkonnosť než Jacobiho metóda. To čiastočne vyplýva z horšej práce s pamäťou. Jacobiho algoritmus má vďaka jednému prechodu maticou 2-krát koncentrovanejší prístup do pamäti než Gauss-Seidel, vďaka čomu častejšie nachádza dáta v cache. Hlavnou príčinou nižšej výkonnosti je vektorizácia, ktorá poskytla len $\sim 30\%$ nárast výkonnosti. Nízka úroveň kvality vektorizácie je spôsobená prístupom do pamäti s rozostupom.



Obrázok 3.21: Histogram porovnania rýchlosti výpočtu LaPlaceovej rovnice plne optimalizovanými verziami algoritmov Jacobi a Gauss-Seidel. Časová os má logaritmickú mierku so základom 10. Benchmark bol spustený s parametrami $E = 0.01, I = 100000$. Histogram ukazuje rádovo nižšiu dobu výpočtu algoritmom Gauss-Seidel napriek oveľa lepšej úrovni optimalizácie Jacobiho metódy. Dôvodom je výrazne lepšia úroveň konvergencie výpočtu Gauss-Seidelovou metódou.

Kapitola 4

Záver

Hlavným cieľom tejto práce bolo ukázať dôležitosť optimalizácie kódu v aplikáciach určených pre moderné procesory. V úvodnej časti práce sa nachádza teoretický základ popisujúci súčasné architektúry procesorov a najdôležitejšie hardwarové prvky, ktoré majú vplyv na výkonnosť softwarovej implementácie. Znalosť týchto teoretických princípov je využiteľná aj v tých najjednoduchších aplikáciach a vedie k oveľa efektívnejšej implementácii. Vyššia efektivita znamená kratšiu výpočetnú dobu, ktorá má priamy dopad na užívateľskú skúsenosť, cenu aplikácie alebo aj celkovú použiteľnosť v praxi.

Práca je zameraná na techniky asistovanej paralelizácie a vektorizácie, ktoré poskytuje štandard OpenMP 4.0. V praktickej časti práce je tento štandard testovaný na jednoduchých optimalizačných problémoch ako je skalárny súčin vektorov, ale aj na optimalizačne náročnejších problémoch.

Jednoduché benchmarky otestovali základné vlastnosti paralelizácie a vektorizácie operácií nad súvisle uloženými kolekciami dát, ktoré sú spracovávané počítaným cyklom.

Benchmark algoritmu mergesort ukázal možný spôsob optimalizácie algoritmov, ktoré nepracujú nad dátovými štruktúrami súvislým spôsobom. Navyše sa otestoval prediktor skokov a vplyv usporiadania prvkov na úroveň zlej predikcie, ktorá pre náhodne usporiadané pole dosahovala v závislosti od veľkosti poľa hodnoty v rozmedzí 10 až 20%.

Benchmark vyhľadávania vo výškovo vyváženom binárnom strome otestoval činnosť TLB. Dôležitým poznatkom získaným z tohto benchmarku je, že priemerný počet výpadkov TLB na vyhľadanie prvku v strome nie je priamo závislý len na výške stromu, ale aj na veľkosti uzlov daného stromu. TLB dosahovalo až dvojnásobne väčší počet výpadkov pre stromy s uzlami o veľkosti 1KB, ako pre stromy s uzlami o veľkosti 24 bytov. Na základe výsledku benchmarku doporučujem neukladať do vyhľadávacích štruktúr celé objekty, ale len vyhľadávací kľúč a referenciu na objekt.

Benchmark násobenia matíc otestoval algoritmy, ktoré využívali rôzne optimalizačné techniky ako preusporiadanie prvkov a blokovanie za účelom zlepšenia využitia cache. Výsledky testovania ukázali dôležitosť správneho návrhu algoritmu. Zatiaľ čo najpoužívanejšie naivné riešenie dosahovalo maximálnu výkonnosť len 60 GFLOPS, jednoduchá úprava transpozíciou alebo zmenou prístupového vzoru dosiahovali výkonnosť až 300 GFLOPS. V rámci benchmarku bola porovnaná implementácia blokového algoritmu s profesionálne vytvorenou implementáciou z knižnice MKL. Napriek oveľa nižším nákladom na vývoj použitej blokovej implementácie bola dosahovaná výkonnosť v priemere len 8-krát nižšia než výkonnosť MKL implementácie.

V poslednom benchmarku boli testované spôsoby optimalizácie numerického riešenia diferenciálnych rovníc. Riešenie pomocou Jacobiho metódy nepredstavovalo optimalizačne

náročný problém a dosahovalo výborný nárast výkonnosti až na 180 GFLOPS. Gauss-Seidelova metóda si vďaka svojej dedične sekvenčnej povahe vynútila použitie tzv. farbenia prvkov matice. Rozdelenie prvkov matice do dvoch skupín umožnilo paralelizáciu aj vektorizáciu. Vektorizácia však v tomto prípade nedosahovala vysokú úroveň efektivity a riešenie po optimalizácii dosiahlo výkonnosť len 65 GFLOPS. Oveľa lepšia úroveň konvergencie Gauss-Seidelovej metódy sa v závere prejavila na celkovej dobe výpočtu, kedy k výsledku konvergovala rýchlejšie o približne 1,5 rádu.

Vďaka jednoduchosti používania a nepopierateľnému prínosu, by mal byť štandard OpenMP 4.0 súčasťou portfólia každého Fortran, C a C++ programátora. Netreba zabúdať, že optimalizácia kódu a optimalizácia algoritmu sú dva rozdielne princípy, ktoré si častokrát môžu odporovať, ale len ich vhodným spojením vznikne tá najlepšia možná implementácia. Pri návrhu akéhokoľvek algoritmu, ktorý bude pracovať s veľkým množstvo dát je nutné brať ohľad na usporiadanie dát v pamäti, spôsob akým sa k dátam pristupuje a ako to bude vplývať na využitie cache. Preusporiadanie dát v pamäti môže znieť kontraproduktívne, ale vo výsledku môže získať niekoľkonásobný nárast výkonnosti.

Vďaka tejto práci som získal neoceniteľné znalosti optimalizačných techník. Za najväčší prínos práce považujem implementáciu blokového algoritmu maticového súčinu, ktorá vykazuje lepšie vlastnosti než akákoľvek iná C/C++ implementácia tohto algoritmu, s ktorou som sa stretol. Práca môže vďaka obšírnej charakteristike štandardu OpenMP 4.0 a dostatku praktických príkladov slúžiť aj ako manuál.

Ďalší vývoj práce môže spočívať v implementácií a analýze výkonnosti iných optimalizačných techník spojených so štandardom OpenMP a v popise nadchádzajúcich verzí OpenMP.

Literatúra

[1] Agner, F.: *Optimizing software in C++ [online]*. Technická zpráva, Technical University of Denmark, 2015.
URL http://www.agner.org/optimize/optimizing_cpp.pdf

[2] Annavaram, M.; Grochowski, E.: *Energy per Instruction Trends in Intel® Microprocessors [online]*. Technická zpráva, Microarchitecture Research Lab, Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054.
URL
http://www.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf

[3] Blaise, B.; Livermore, L.: *Tutorial OpenMP [online]*.
URL https://computing.llnl.gov/tutorials/openMP/

[4] Duffy, A.: *Creating and Using a Red-Black Matrix [online]*. Technická zpráva, 2010.
URL http://computationalmathematics.org/topics/files/red_black.pdf

[5] Evers, M.: *Improving Branch Prediction by Understanding Branch Behavior [online]*. Technická zpráva, University of Michigan, 2000.
URL https://www.eecs.umich.edu/techreports/cse/99/CSE-TR-417-99.pdf

[6] Hansen, P. B.: *Numerical Solution of Laplace's Equation [online]*. Technická zpráva, Electrical Engineering and Computer Science Technical Reports, 1992.
URL http://surface.syr.edu/cgi/viewcontent.cgi?article=1160&context=eecs_techreports

[7] Kolektív autorov: *Allinea MAP [online]*.
URL http://www.allinea.com/products/map

[8] Kolektív autorov: *Dokumentácia IT4Innovations [online]*.
URL https://docs.it4i.cz/get-started-with-it4innovations

[9] Kolektív autorov: *Gnuplot homepage [online]*.
URL http://www.gnuplot.info/

[10] Kolektív autorov: *Intel Vtune Amplifier [online]*.
URL https://software.intel.com/en-us/intel-vtune-amplifier-xe

[11] Kolektív autorov: *PAPI documentation [online]*.
URL http://icl.cs.utk.edu/projects/papi/wiki/Main_Page

[12] Kolektív autorov: *PBS Professional User's Guide [online]*.
URL http://www.pbsworks.com/pdfs/PBSUserGuide13.0.pdf

[13] Kolektív autorov: *User and Reference Guide for the Intel C++ Compiler 15.0 [online].*
URL https://software.intel.com/en-us/compiler_15.0_ug_c

[14] Kolektív autorov: *Performance Essentials with OpenMP 4.0 Vectorization [online].* 2013.
URL https://software.intel.com/en-us/articles/performance-essentials-with-openmp-40-vectorization

[15] Kolektív autorov: *7-Zip LZMA Benchmark [online].* 2016.
URL http://www.7-cpu.com/utils.html

[16] Kolektív autorov: *Dynamic random-access memory [online].* 2016.
URL https://en.wikipedia.org/wiki/Dynamic_random-access_memory

[17] Kolektív autorov: *Static random-access memory [online].* 2016.
URL https://en.wikipedia.org/wiki/Static_random-access_memory

[18] Kolektív autorov: *OpenMP Application Program Interface [online].* July 2013.
URL http://www.openmp.org

[19] Oliker, L.; Canning, A.; Carter, J.; aj.: *Scientific Computations on Modern Parallel Vector Systems [online].* Technická zpráva, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, 2004.
URL http://crd-legacy.lbl.gov/~oliker/papers/SC04.pdf

[20] Orság, F.: *Studijní opora k předmětu pokročilé assemblery.* 2006.

[21] Wikipedia: *Moore's law [online].* 2016.
URL https://en.wikipedia.org/wiki/Moore%27s_law

Prílohy

Zoznam príloh

A Obsah CD **45**

B Kompletný zoznam OpenMP 4.0 runtime funkcií **46**

C Kompletný zoznam OpenMP 4.0 premenných prostredia **48**

D PBS Pro **49**

E Zoznam dôležitých optimalizačných prepínačov Intel C/C++ prekladača **50**

F Implementácia blokového algoritmu maticového súčinu **52**

G Maticový súčin - cache **53**

H Odvodenie vzorca pre numerický výpočet LaPlaceovej rovnice **55**

Príloha A

Obsah CD

Priložené CD obsahuje:

- Makefile ku zdrojovým súborom
- priečinok `src` obsahujúci všetky zdrojové súbory benchmarkov
- priečinok `results` obsahujúci výsledky benchmarkov
- priečinok `thesis` obsahujúci zdrojové súbory písomnej práce pre LaTeX

Príloha B

Kompletný zoznam OpenMP 4.0 runtime funkcií

`omp_set_num_threads`	Nastavenie žiadaného počtu vlákien v par. sekciách
`omp_get_num_threads`	Počet vlákien v par. sekcii
`omp_get_max_threads`	Maximálny počet vlákien v par. sekcii
`omp_get_thread_num`	Číslo volajúceho vlákna
`omp_get_num_procs`	Počet prístupných procesorov v čase volania
`omp_in_parallel`	*True* ak volané z par. sekcie
`omp_set_dynamic`	Nastavenie dynamického nastavovania počtu vlákien za behu
`omp_get_dynamic`	Vráti nastavenie dynamického nastavovania počtu vlákien za behu
`omp_set_nested`	Nastavenie vnoreného paralelizmu
`omp_get_nested`	Vráti nastavenie vnoreného paralelizmu
`omp_set_schedule`	Nastavenie rozvrhu ak je v klauzulách výber pomocou `runtime`
`omp_get_schedule`	Vráti nastavenie rozvrhu
`omp_get_thread_limit`	Limit OpenMP vlákien
`omp_set_max_active_levels`	Nastavenie maximálneho počtu úrovní vnoreného paralelizmu
`omp_get_max_active_levels`	Vráti nastavenie maximálneho počtu úrovní vnoreného paralelizmu
`omp_get_level`	Počet úrovní vnoreného paralelizmu v okamihu volania
`omp_get_ancestor_thread_num`	Vráti ID *master* vlákna pre úroveň vnoreného paralelizmu v okamihu volania
`omp_get_team_size`	Vráti počet vlákien vo vnorenej par. sekcii
`omp_get_active_level`	Počet aktívnych vnorených par. sekcií
`omp_in_final`	*True* ak volané z finálnej vnorenej par. sekcie

Tabuľka B.1: Funkcie nastavujúce OpenMP prostredie

`omp_get_cancellation`	Nastavenie správania `cancel` konštrukcie
`omp_get_proc_bind`	Vráti nastavenie rozdelovania vlákien medzi procesory
`omp_set_default_device`	Nastavenie štandardného cieľového zariadenia
`omp_get_default_device`	Vráti nastavenie štandardného cieľového zariadenia
`omp_get_num_devices`	Počet cieľových zariadení
`omp_get_num_teams`	Počet skupín vlákien v súčasnom `team` regióne
`omp_get_team_num`	Číslo skupiny volaného vlákna
`omp_is_initial_device`	*True* ak volané z vlákna vykonávaného na hosťovskom zariadení

Tabuľka B.2: Nové funkcie v **OpenMP 4.0** nastavujúce prostredie

`omp_get_wtime`	Ubehnutý čas v sekundách
`omp_get_wtick`	Presnosť časovača (sekundy medzi tikmi hodín)

Tabuľka B.3: OpenMP časovacie funkcie

`omp_init_lock`	Vytvorenie bežného zámku
`omp_init_nest_lock`	Vytvorenie zámku, ktorý môže byť nastavený rovnakým vláknom viac krát
`omp_destroy_lock`	Zničenie bežného zámku
`omp_destroy_nest_lock`	Zničenie vnoriteľného zámku
`omp_set_lock`	Nastavenie bežného zámku
`omp_set_nest_lock`	Nastavenie vnoriteľného zámku
`omp_unset_lock`	Zrušenie nastavenia bežného zámku
`omp_unset_nest_lock`	Zrušenie nastavenia vnoriteľného zámku
`omp_test_lock`	Otestovanie nastavenia bežného zámku
`omp_test_nest_lock`	Otestovanie nastavenia vnoriteľného zámku

Tabuľka B.4: OpenMP funkcie nastavujúce zámky

Príloha C

Kompletný zoznam OpenMP 4.0 premenných prostredia

`OMP_DYNAMIC`	Nastavenie dynamického počtu vlákien
`OMP_MAX_ACTIVE_LEVELS`	Nastavenie maximálneho počtu actívnych úrovní vnoreného paralelizmu
`OMP_NESTED`	(De)aktivácia vnoreného paralelizmu
`OMP_NUM_THREADS`	Počet vlákien, o ktoré OpenMP požiada systém v paralelnej sekcii.
`OMP_PROC_BIND`	Nastavenie rozloženia vlákien medzi procesory
`OMP_SCHEDULE`	Nastavenie štandardného rozvrhu pri použití `schedule(runtime)`
`OMP_STACKSIZE`	Veľkosť extra pamäťového priestoru na zásobníku pre každé vlákno. Umožňuje včasné odhalenie potenciálneho pretečenia zásobníku a teda pádu systému.
`OMP_THREAD_LIMIT`	Nastavenie limitu OpenMP vlákien
`OMP_WAIT_POLICY`	Definuje spôsob čakania vlákien. V prípade aktívneho čakania vlákno opakovane testuje podmienku pokračovania v činnosti. Pasívny spôsob čakania uspí vlákno až kým nebude môcť pokračovať. Aktívne čakanie je vhodné pri krátkej čakacej dobe, pasívne naopak

Tabuľka C.1: OpenMP premenné prostredia

`OMP_CANCELLATION`	(De)aktivácia `cancel` konštrukcie
`OMP_DEFAULT_DEVICE`	Nastavenie štandardného zariadenia
`OMP_DISPLAY_ENV`	(De)aktivácia zobrazenia nastavení OpenMP prostredia na začiatku behu aplkácie
`OMP_PLACES`	Nastavenie OpenMP miest prístupných OpenMP prostrediu. Môže byť *threads/cores/sockets*

Tabuľka C.2: Nové premenné prostredia v **OpenMP 4.0**

Príloha D

PBS Pro

PBS umožňuje alokovať zdroje férovo pomocou frontov, do ktorých sa môže užívateľ zapísať podľa potreby (náročnosť aplikácie, potrebný počet jadier...).

`qexp`	Expresný front pre krátke úlohy
`qprod`	Produkčný front pre bežné úlohy
`qlong`	Front pre náročné úlohy
`qmpp`	Front pre masívne náročne úlohy
`qfat`	Front sprístupňujúci *SMP UV2000*
`qfree`	Front poskytujúci práve voľné zdroje
`qviz`	Front určený na vizualizáciu pomocou OpenGL

Tabuľka D.1: PBS fronty s krátkym popisom. Podrobnejšia špecifikácia je dostupná v PBS Professional užívateľskom manuály [12]

Užívateľ pridáva úlohy do frontov pomocou príkazu `qsub` typicky ako:

```
qsub -A {project_ID} -q {queue}
     -l select={nodes}:ncpus={cores},walltime={timestamp}
     {jobscript}
```

Na vytvorenie interaktívnej úlohy sa namiesto `{jobscript}` používa prepínač `-I`. V prípade, že úloha využíva moduly s grafickým rozhraním ako Allinea MAP (viď. 2.5.2), je nutné použiť prepínač `-X`. Ak sa na meranie využíva profiler Intel VTune Amplifier, je nutné špecifikovať požadovanú verziu už pri alokácii zdrojov.

```
qsub -q {queue} -A {project_ID}
     -l select={nodes},vtune=2016_update1
```

Príloha E

Zoznam dôležitých optimalizačných prepínačov Intel C/C++ prekladača

Kompletná dokumentácia Intel C++ prekladača [13]

`-openmp`	Povolenie OpenMP
`-lpapi`	Nalinkovanie PAPI
`-mkl`	Povolenie knižnice MKL

Tabuľka E.1: Prepínače na sprístupnenie knižníc a štandardov

`-O0`, `-O1`, `-O2`, `-O3`	Úroveň automatickej optimalizácie vrátane vektorizácie (`-O0` vypne)
`-vec`, `-no-vec`	Nastavenie automatickej vektorizácie
`-restrict`, `-no-restrict`	Povolenie významu kľúčového slova `restrict`
`-msse`, `-msse2`, `-msse3`, `-msse4.1`, `-msse4.2`, `-mavx`	Povolenie rozšírení inštrukčnej sady. Povolenie novšej sady povoluje aj všetky staršie.
`-xHost`	Vygenerovanie inštrukcií najnovšou inštrukčnou sadou dostupnou na danom procesore
`-align`, `-no-align`	Nastavenie prirodzeného zarovnávania premenných a polí. Štandardne sú dáta zarovnané podľa gcc modelu (na 4 byty)
`-ansi-alias`, `-no-ansi-alias`	Povolenie používania ANSI aliasing pravidiel pri optimalizácii

Tabuľka E.2: Prepínače ovplyvňujúce vektorizáciu. Podčiarknuté hodnoty sú štandardné

`-opt-report=[úroveň]`	Výpis optimalizačnej správy na `stderr` s požadovanou úrovňou detailnosti výpisu.
`-opt-report-file=[názov súboru]`	Určenie názvu súboru kde bude uložený výpis (môže byť `stdout`)
`-opt-report-phase=[fáza]`	Definovanie použitej optimizačnej fázy počas ktorej bude výpis generovaný
`-opt-report-routine=\ [reťazec]`	Optimalizačná správa bude vygenerovaná pre rutiny obsahujúce vo svojom názve zadaný reťazec

Tabuľka E.3: Debugovacie prepínače

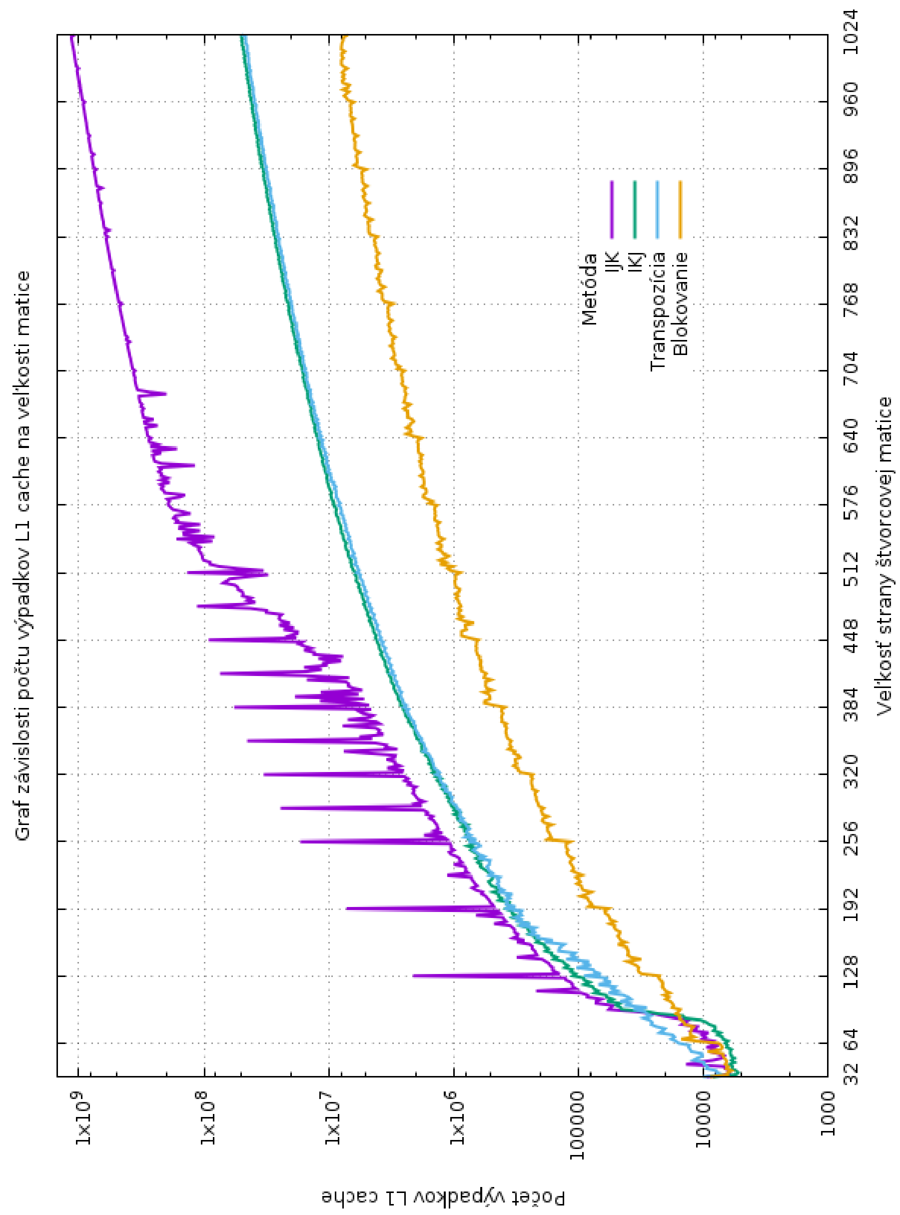
Príloha F

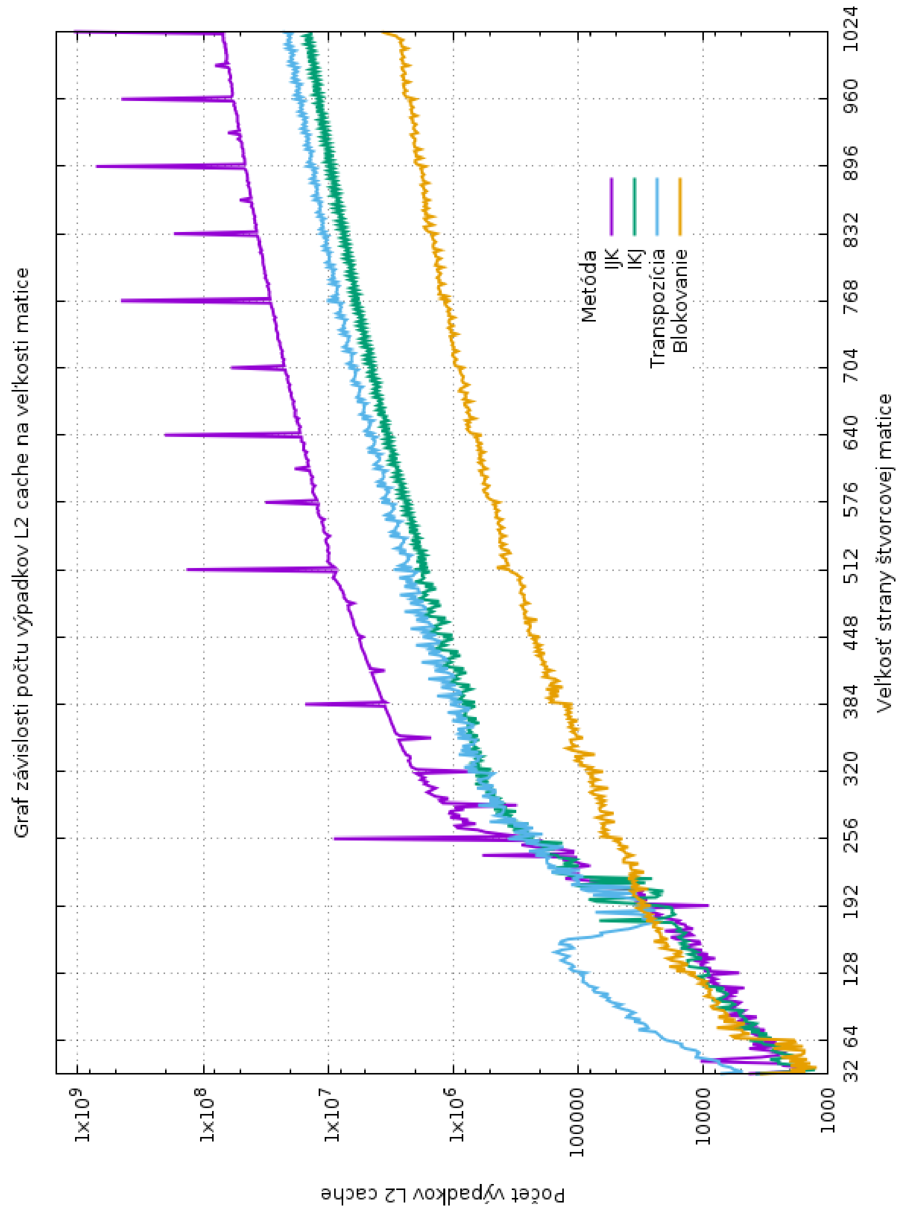
Implementácia blokového algoritmu maticového súčinu

```
1  int i, j, k, ii, jj, kk, x, y;
2  float TC[BS*BS], TA[BS*BS], TB[BS*BS];   // BS = Veľkosť bloku
3
4  #pragma omp parallel for default(shared) collapse(3) \
5   schedule(static) private(TC, TA, TB, j, x, y, k, ii, jj, kk)
6  for(i = 0; i < N; i += BS) {
7    for(j = 0; j < N; j += BS) {
8      for(k = 0; k < N; k += BS) {
9        int i_BS = (N - i < BS) ? N - i : BS;
10       int j_BS = (N - j < BS) ? N - j : BS;
11       int k_BS = (N - k < BS) ? N - k : BS;
12
13       for(x = 0; x < i_BS; ++x)           // Načítaj blok z A
14       for(y = 0; y < k_BS; ++y)
15         TA[x * BS + y] = A[(i + x) * N + k + y];
16
17       for(x = 0; x < k_BS; ++x)           // Načítaj blok z B
18       for(y = 0; y < j_BS; ++y)
19         TB[y * BS + x] = B[(k + x) * N + j + y];
20
21       for(ii = 0; ii < i_BS; ++ii)        // Vypočítaj SGEMM
22       for(jj = 0; jj < j_BS; ++jj) {
23         float temp = 0.0f;
24         #pragma omp simd reduction(+:temp)
25         for(kk = 0; kk < k_BS; ++kk)
26           temp += TA[ii * BS + kk] * TB[jj * BS + kk];
27         TC[ii * BS + jj] = temp;}
28
29       for(x = 0; x < i_BS; ++x)           // Ulož výsledok
30       for(y = 0; y < j_BS; ++y)
31          C[(i + x) * N + j + y] += TC[x * BS + y];
32 } } }
```

Príloha G

Maticový súčin - cache





Grafy používajú logaritmickú mierku na zvislej osi. Na grafoch je vidieť efektivitu využívania L1 a L2 cache. Algoritmus IJK je už podľa týchto grafov zjavne nestabilný. Grafy ukazujú veľký nárast L1 aj L2 *cache miss* pre hodnotu 512. Dosiahnutá výkonnosť IJK pre túto hodnotu je skutočne takmer 2-krát nižšia než pre 513 alebo 511. Ostatné algorimy si udržujú stabilitu pričom pomer počtu *cache miss* medzi jednotlivými algoritmami zodpovedá pomeru ich výkonností.

Príloha H

Odvodenie vzorca pre numerický výpočet LaPlaceovej rovnice

Zdroj viď. [6].

Nakoľko hodnota potenciálu sa v každej časti plochy prakticky nemení, môžeme predpokladať, že hodnota potenciálu v ľubovoľnej časti je odvoditeľná ako súčet jej obkolesujúcich častí. Pomocou Taylorovho rozvoja môžeme určiť hodnoty susedných častí plochy vľavo a vpravo nasledovne:

$$U(x+\Delta x, y) = U(x,y) + \Delta x \frac{\partial}{\partial x} U(x,y) + \Delta x^2 \frac{\partial^2}{\partial^2 x} U(x,y) + \Delta x^3 \frac{\partial^3}{\partial^3 x} U(x,y) + ...$$
$$U(x-\Delta x, y) = U(x,y) - \Delta x \frac{\partial}{\partial x} U(x,y) + \Delta x^2 \frac{\partial^2}{\partial^2 x} U(x,y) - \Delta x^3 \frac{\partial^3}{\partial^3 x} U(x,y) + ...$$
$$U(x+\Delta x, y) + U(x-\Delta x, y) = 2U(x,y) + \Delta x^2 \frac{\partial^2}{\partial^2 x} U(x,y)$$

Pre časti plochy nad a pod platí:

$$U(x, y+\Delta y) = U(x,y) + \Delta y \frac{\partial}{\partial y} U(x,y) + \Delta y^2 \frac{1}{2} \frac{\partial^2}{\partial^2 y} U(x,y) + \Delta y^3 \frac{1}{6} \frac{\partial^3}{\partial^3 y} U(x,y) + ...$$
$$U(x, y-\Delta y) = U(x,y) - \Delta y \frac{\partial}{\partial y} U(x,y) + \Delta y^2 \frac{1}{2} \frac{\partial^2}{\partial^2 y} U(x,y) - \Delta y^3 \frac{1}{6} \frac{\partial^3}{\partial^3 y} U(x,y) + ...$$
$$U(x, y+\Delta y) + U(x, y-\Delta y) = 2U(x,y) + \Delta y^2 \frac{\partial^2}{\partial^2 y} U(x,y)$$

Potom je celkový súčet rovný:

$$U(x+\Delta x, y) + U(x-\Delta x, y) + U(x, y+\Delta y) + U(x, y-\Delta y) = 4U(x,y) + \Delta x^2 \frac{\partial^2}{\partial^2 x} U(x,y) + \Delta y^2 \frac{\partial^2}{\partial^2 y} U(x,y)$$

$$\boldsymbol{U(x,y) = \frac{U(x+\Delta x, y) + U(x-\Delta x, y) + U(x, y+\Delta y) + U(x, y-\Delta y)}{4}}$$