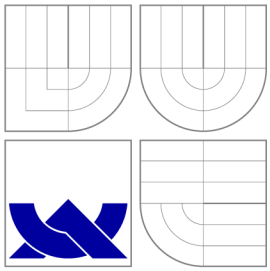
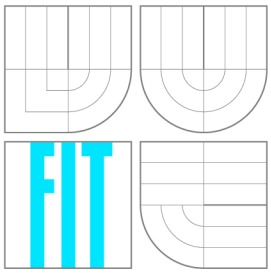


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

VÝVOJ PARALELNÍCH APLIKACÍ S INTEL THREADING TOOLS

MULTITHREADED APPLICATION DEVELOPMENT WITH INTEL THREADING TOOLS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LADISLAV VADKERTI
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. VÁCLAV DVOŘÁK, DrSc.
SUPERVISOR

BRNO 2007

Vývoj paralelních aplikací s Intel Threading Tools

Parallel Application Development with Intel Threading Tools

Vedoucí:
Dvořák Václav, prof. Ing., DrSc., UPSY FIT VUT

Oponent:
Jaroš Jiří, Ing., UPSY FIT VUT

Přihlášen:
Vadkerti Ladislav, Bc.

Zadání:

1. Seznamte se s Intel threading tools (Intel Thread Checker, Intel Thread Profiler, and VTune Performance Analyzer)
2. Napište a odlaďte program např. pro násobení matic pomocí Intel threading tools.
3. Proveďte program na multiprocesoru Sun nebo na dvoujádrovém procesoru Intel.
4. Zhodnoťte zkušenosti s prací s Intel threading tools a sepište stručný manuál pro začínající uživatele.
5. Vyjádřete se k užitečnosti zakoupení multilicence pro výuku paralelního programování.
6. Vyjádřete se k případným jiným nástrojům pro programování s vlákny, pokud budou k dispozici.

Část požadovaná pro obhajobu SP:
Splnění prvních 2 bodů zadání.

Kategorie:
Uživatelská rozhraní

Implementační jazyk:
C

Operační systém:
Unix

Návrhová metodologie:
Intel threading tools

Komerční software:
Intel threading tools 2.2

Volně šířený software:
OpenMP

Literatura:

- Architektura a programování paralelních systémů. Skripta FIT VUT v Brně, VUTIUM 2004.
- Dokumentace Intel threading tools 2.2

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Dnešním trendem v návrhu mikroprocesorů je zvyšování počtu výkonných jader na jednom čipu. Zvyšování taktovací frekvence dosáhlo svých limitů způsobených rostoucí energetickou spotřebou. Tento trend přináší nové možnosti pro softwarové vývojáře, kteří mohou využít skutečného paralelizmu ve vykonávání více vláken v rámci aplikace. Ale současný běh vláken také přináší nové problémy, které se při vývoji sekvenčních programů nemusely řešit. Správně navržená aplikace může použitím více vláken dosáhnout zlepšení výkonu lepším využitím hardwarových prostředků. Na druhou stranu, nesprávné použití vláken může vést k degradaci výkonu, nepředvídatelnému chování a chybovým stavům, které jsou těžko řešitelné. Z tohoto důvodu firma Intel vyvinula sadu nástrojů, které mají napomáhat vývojářům analyzovat výkon a detekovat chyby v interakci mezi vlákny. Tato práce se zaměřuje na možnosti použití těchto nástrojů při vývoji vícevláknových aplikací.

Klíčová slova

paralelní programování, vlákna, ladění výkonu, vývojové nástroje, násobení matic

Abstract

Today's trend in microprocessor design is increasing the number of execution cores within one single chip. Increasing the processor's clock speed reached its limit with growing power consumption. This trend brings new opportunities to software developers, as they can take advantage of real multithreading in their applications. But a lot of new problems to solve appear with threading compared to sequential programming. With proper design, threading can enhance performance by making better use of hardware resources. However, the improper use of threading can lead to performance degradation, unpredictible behavior, or error conditions that are difficult to solve. For this reason Intel developed a suite of tools, that can help software developers to analyze performance and detect coding errors in thread interactions. This thesis focuses on the examination of ways that this tools can be used in multithreaded application development.

Keywords

parallel programming, multithreading, performance tuning, development tools, matrix multiplication

Citace

Ladislav Vadkerti: Vývoj paralelních aplikací
s Intel Threading Tools, diplomová práce, Brno, FIT VUT v Brně, 2007

Vývoj paralelních aplikací s Intel Threading Tools

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Václava Dvořáka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ladislav Vadkerti
22. května 2007

Obsah

Kapitola 1

Úvod

V súčasnosti nastupuje nový trend v návrhu procesorových architektúr. Intel, IBM, Sun a AMD predstavili mikroprocesory s niekoľkými jadrami na jednom čipe. Týmito procesormi boli osadené pracovné stanice, servery, herné konzoly. Predbežné plány týchto firiem tiež predpovedajú, že toto je iba začiatok. Skôr než honba za prvenstvom v dosiahnutí taktovacej frekvencie procesorov 10 gigahertz sa výrobcovia mikroprocesorov predbiehajú v integrovaní čo najväčšieho počtu jadier na jediný čip.

Skutočnosť, že sa mikroprocesorový priemysel vydal týmto smerom, znamená nové príležitosti pre softvérových vývojárov. Predošlé hardvérové platformy predstavovali z pohľadu programátora sekvenčný programovací model. Operačné systémy simulovali paralelné vykonávanie úloh využitím vysokého výkonu výpočtových systémov. Výsledkom bola ilúzia súčasného behu viacerých vlákien. Moderné viacjadrové architektúry však predstavujú skutočnú paralelnú výpočtovú platformu. Softvérovým vývojárom sa tak ponúkajú nové možnosti návrhu a implementácie softvéru. Tieto nové možnosti však so sebou prinášajú aj nové problémy ako pamäťová konzistencia alebo blokovanie vlákien, ktorých si programátor musí byť vedomý. Paralelné programovanie ale nie je čerstvo vznikajúca disciplína. Je prítomné už niekoľko desaťročí. Za tú dobu boli identifikované najčastejšie problémy súvisiace s paralelným vykonávaním vlákien a tiež navrhnuté vzory pre riešenie týchto problémov.

Cieľom použitia viacerých vlákien na spracovanie úlohy je najčastejšie dosiahnutie zvýšenia výkonu aplikácie. To však vyžaduje dômyselný návrh rozloženia zaťaženia medzi niekoľko súčasne vykonávaných vlákien. Vyladenie aplikácie na najvyšší výkon a nedopustiť sa pri tom chýb sú preto jedným z priorít softvérových vývojárov využívajúcich paralelizmus ponúkaný v moderných mikroprocesorových architektúrach. Na zjednodušenie a zefektívnenie ich práce preto vzniklo niekoľko softvérových nástrojov, z ktorých dvom sa venuje táto práca. Ide o výkon aplikácie analyzujúci nástroj Intel Thread Profiler a problémy, spojené so súčasným behom vlákien, identifikujúci nástroj Intel Thread Checker. Ich využitím som sa pokúsil o dosiahnutie čo najvyššieho výkonu pri riešení úlohy násobenia dvoch matíc ľubovolných rozmerov na viacjadrovom procesore.

Druhá kapitola tejto práce sa venuje kategorizácii výpočtových architektúr a zaradeniu súčasných najmodernejších mikroprocesorových architektúr do tohto systému. Načrtáva evolúciu paralelizmu v procesoroch a zachytáva súvislosti medzi týmto vývojom a problémami paralelného programovania. V závere ponúka kľúč k pochopeniu prínosu zvyšovania výkonu paralelizovaním úloh.

Tretia kapitola vysvetľuje základné pojmy viacvláknového spracovania. Definuje pojem vlákna a popisuje ho z pohľadu troch úrovní, na ktorých sa môže vyskytnúť. Vy-

kresľuje ho z pohľadu používateľskej aplikácie, operačného systému a hardvéru. Popisuje deje sprevádzajúce vznik vlákna. Vymenováva výzvy, ktorým musí programátor čeliť pri návrhu aplikácie a predkladá sortiment mechanizmov, ktoré má k dispozícii na riešenie.

Štvrtá kapitola ponúka prehľad funkcií a základných princípov nástrojov na analýzu výkonu a identifikáciu problémov aplikácií pracujúcich s viacerými vláknami Intel Thread Profiler a Intel Thread Checker. Pridáva porovnanie niekoľkých ďalších programov plniacich rovnaký alebo podobný účel ako softvér od firmy Intel.

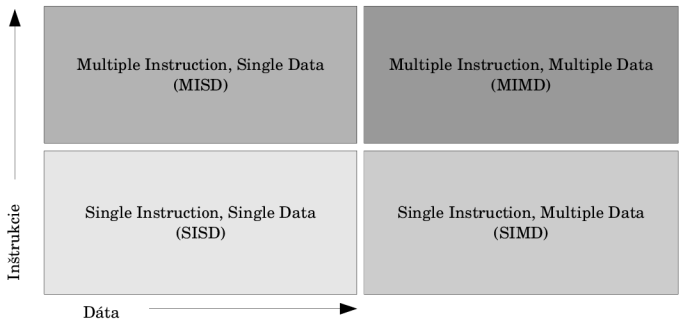
Piata kapitola popisuje spôsob riešenia úlohy násobenia matíc na viacprocesorovom systéme. Cieľom úlohy je maximalizácia výkonu tejto operácie za použitia moderných analyzačných nástrojov. Keďže ide o operáciu náročnú na pamäťovú priepustnosť systému, hlavný zreteľ je kladený na prácu s vyrovnávacími pamäťami procesorov. Na záver hodnotí úspešnosť dosiahnutia vytýčeného cieľa a to porovnaním výkonových výsledkov dvoch variant algoritmu – optimalizovaného a zjednodušeného.

V prílohe sa nachádza stručný návod na použitie nástrojov softvérového balíka Intel Threading Tools, ktorý krok za krokom prevedie neskúseného používateľa príkladmi, demonštrujúcimi možnosti týchto programov. Po absolvovaní tohto jednoduchého kurzu bude schopný vytvoriť a nastaviť projekt, vykonať analýzu výkonu, prípadne synchronizačných problémov, a upraviť ukážkové zdrojové kódy tak, aby pracovali efektívnejšie, prípadne bez chýb plynúcich zo súčasného behu viacerých vlákien.

Kapitola 2

Paralelné architektúry

K dosiahnutiu skutočného paralelného vykonávania programu je potrebné, aby aj hardvérová platforma podporovala simultánne vykonávanie viacerých vlákien. Obecne povedané, počítačové architektúry je možné rozdeliť podľa dvoch rozdielnych kritérií. Prvým kritériom je počet inštrukčných tokov, ktoré je počítačová architektúra schopná spracovávať v jednom bode času. Druhým kritériom je počet dátových tokov, ktoré môžu byť spracovávané v jednom bode času. Týmto spôsobom je možné opísať spôsob spracovávania inštrukcií a dát každého výpočtového systému. Tento klasifikačný systém sa nazýva Flynnova taxonómia [7] [3] a rozdeľuje výpočtové architektúry do štyroch kategórií:



Obrázek 2.1: Flynnova taxonómia

- Single Instruction, Single Data (SISD) stroj je klasický sekvenčný počítač bez podpory paralelizmu. Vykonáva jednu inštrukciu za takt, v ktorej spracováva jediný dátový prúd. Typickými predstaviteľmi tejto triedy výpočtových systémov sú prvé počítače IBM PC, staré mainframe počítače alebo populárne 8-bitové domáce počítače Commodore 64.
- Multiple Instruction, Single Data (MISD) stroj je schopný spracovať väčší počet inštrukcií za jeden takt, ktorý spracováva jeden dátový tok. Keďže vo väčšine prípadov je viac inštrukcií využívaný na spracovanie väčšieho počtu dátových tokov, tento model architektúry je skôr teoretický a v praxi nepoužívaný.
- Single Instruction, Multiple Data (SIMD) stroj môže spracovávať jeden tok inštrukcií,

ktorý spracováva viacero dátových tokov súčasne. Táto trieda výpočtových systémov je užitočná v obecnom spracovaní digitálnych signálov, spracovaní obrazu, multimediálnych aplikáciách ako audio a video. Väčšina súčasných počítačov nejakým spôsobom implementuje SIMD inštrukčnú sadu. Intel procesory obsahujú MMX, Streaming SIMD Extension (SSE), ktoré sú schopné spracovať viac jednotiek informácie v jednom takte procesora, pričom dáta sú uložené v registrovom poli desatinných čísel.

- Multiple Instruction, Multiple Data (MIMD) stroj je schopný spracovávať viac inštrukčných tokov súčasne, pričom každý jeden inštrukčný tok spracováva nezávislý tok dát. Táto trieda predstavuje typický paralelný systém súčasnosti, a aj procesory Intel Core Duo spadajú do tejto kategórie.

Dnešné moderné výpočtové stroje patria buď do kategórie SIMD alebo MIMD, softvérový vývojári tak majú možnosť využiť dátový alebo inštrukčný paralelizmus.

2.1 Paralelizmus v procesoroch

Známy Moorov zákon [3], ktorý tvrdí, že počet tranzistorov v mikročipoch sa zdvojnásobuje každých 18 až 24 mesiacov, platí už viac než 50 rokov. Posledné štyri desaťročia ovplyvňoval návrhárov procesorov. Kým pri spomenutí zvyšovania výkonu sa každému zväčša vybaví zvyšovanie taktovacej frekvencie procesoru, vývojári procesorov sa v poslednej dobe prikláňajú skôr k riešeniam lepšieho využitia jednotlivých jednotiek procesorov. V súvislosti s tým sa v procesoroch objavil paralelizmus na úrovni inštrukcií známe tiež ako vykonávanie inštrukcií mimo poradia. Samotný procesor preusporiada prichádzajúce inštrukcie tak, aby boli jeho jednotky čo najoptimálnejšie využité. Týmto spôsobom sa návrháry mikroprocesorov snažia vylepšiť pomer vykonaných inštrukcií za jeden takt. V prípade vykonávania inštrukcií podľa poradia je pre závislosti medzi nimi veľmi ťažké nájsť inštrukcie vykonateľné súčasne. Preto vykonávanie mimo poradia poprehadzuje ich poradie tak, aby sa využili rôzne procesorové jednotky potrebné pre ich vykonanie. Táto technika je pre softvérových vývojárov úplne transparentná a je podporená priamo hardvérom. Je dôležité ju mať na vedomí pri paralelnom programovaní, pretože môže viesť k niektorým mylným prepokladom.

Postupom času, ako sa softvér vyvíjal, stával sa čoraz vhodnejším na vykonávanie viacerých úloh súčasne. Dnešné serverové aplikácia bežia často vo viacerých vláknach alebo procesoch. Aby bolo možné zaviesť paralelizmus na úrovni vlákien, bolo potrebné si osvojiť niekoľko prístupov v návrhu hardvéru a softvéru.

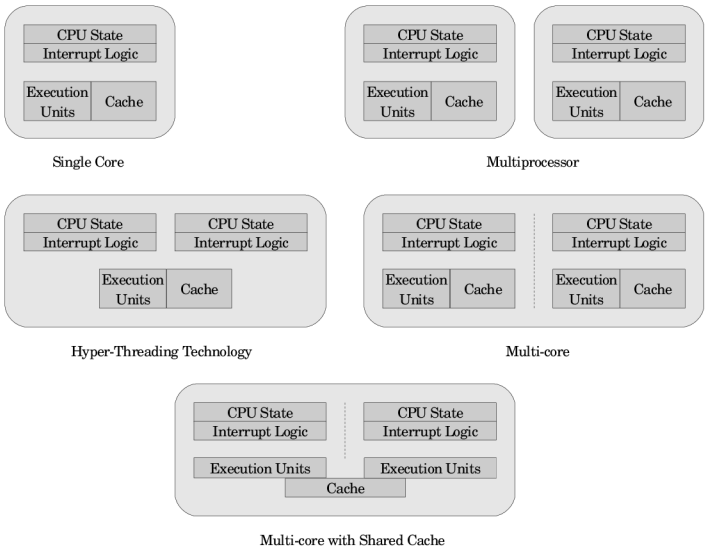
Prvým krokom bolo vyriešenie prístupu aplikácií ku zdieľaným zdrojom výpočtového systému. Tento problém je riešený zavedením preemptívneho vykonávania programov, ktorým je prideľovaný procesorový čas po kvantách. Prekrývaním vykonávania vlákien sa vyriešil aj problém zdržaní spôsobených I/O operáciami. Tento prístup však sám o sebe neumožňuje paralelizmus. Počíta s vykonávaním jednej inštrukcie v jednom časovom okamihu.

Ďalším krokom ku skutočnému viacvláknovému paralelizmu bolo zvyšovanie počtu procesorov vo výpočtovom systéme. Na každom procesore je potom možné vykonávať jeden proces alebo vlákno, čo je už skutočné paralelné vykonávanie inštrukcií.

Jedným z paralelizmu sa blížiacich techník vykonávania vlákien je simultánne viacvláknové vykonávanie (SMT), v podaní Intelu známe ako Hyper-Threading technológia. S jej využitím sa dá dosiahnuť lepšie vyťaženie procesorových jednotiek, pretože vykonávanie programu nie je tak často prerušované ako na procesoroch bez tejto technológie.

Aby sme dospeli k dôvodom jej vzniku, je potrebné si formálnejšie uvedomiť podstatu vlákna. Vlákno si môžeme predstaviť ako jednotku využitia procesoru. Obsahuje informácie o stave procesoru, ukazovateľ na práve vykonávanú inštrukciu a ďalšie prostriedky ako napr. zásobník. Fyzický procesor je tvorený množstvom rôznych prostriedkov ako stavové registry, vyrovnávacie pamäte, registre prerušení a pod., z ktorých však nie sú potrebné všetky pre definíciu vlákna. Preto je možné vytvoriť dva logické procesory oddelením informácií o stave architektúry a zdieľaním výkonných jednotiek procesoru. Vytvorením dvoch logických procesorov sa dosiahne stav, keď operačný systém môže naplánovať vykonávanie dvoch vlákien, pričom mikroarchitektúra preberá úlohu prepínania vlákien na zdieľaných prostriedkoch. Prepnutie môže byť vyvolané výpadkom vo vyrovnávacej pamäti alebo nesprávnou predikciou skoku.

Ďalším logickým krokom od simultánneho viacvláknového vykonávania je viacjadrový procesor. Namiesto zdieľania vybraných prostriedkov na jednom fyzickom procesore sa výkonné jednotky fyzicky znásobia a na jednom čipe je tak viac výkonných jadier. Jadrá majú svoje vlastné stavy architektúry aj vykonné jednotky. Môžu, ale nemusia zdieľať veľkú čipovú vyrovnávaciu pamäť. Takisto môžu byť kombinované s technológiou SMT, ktorá logicky zdvojnásobí počet procesorov.



Obrázek 2.2: Porovnanie jednojadrovej, viacprocesorovej a viacjadrovej architektúry

2.2 Beh vlákien na jednom vs. viac procesoroch

Programovanie s vláknami nie je úplnou novinkou. Mnohé moderné aplikácie používajú viacero vlákien v rámci jedného procesu už v súčasnosti. Vďaka tomu sa už veľká časť programátorov stretla s konceptom paralelného programovania pomocou vlákien, pravdepodobne však cieleným na jednoprocesorovú platformu. Oproti programovaniu pre viacprocesorové systémy tu však sú niektoré rozdiely:

- Programovanie vo vláknach sa používa prednostne na dosiahnutie vyššieho výkonu aplikácie rozdelením spracovania úlohy medzi jednotlivé procesory. Väčšina aplikácií v súčasnosti využíva viac vlákien na zvýšenie odozvy používateľského rozhrania aplikácie. Namiesto čakania na dokončenie rozpracovanej operácie, napr. načítania dát z databázy, zo súboru alebo dokončenia zložitého výpočtu, sa spracovanie udalostí používateľského rozhrania vykonáva vo zvlášť vyčlenenom vlákne. Tým sa dosiahne rovnomerné rozloženie výkonu aplikácie medzi spracovanie úlohy a odozvy používateľského rozhrania.

  Toto je hlavný limitujúci faktor viacvláknových aplikácií na jednoprocesorových systémoch. Namiesto toho, aby sa vlákna používali na zvýšenie výkonu aplikácie, využíva sa iba prekrývanie vykonávania vlákien na odstránenie časovej odozvy. Namiesto toho na viacprocesorových systémoch je možné vykonávať vlákna simultánne, pričom nemusia čakať na dokončenie vykonávania toho druhého. Týmto spôsobom sa ponúka softvérovým vývojárom možnosť optimalizácie aplikácií pre využitie znásobenia prostriedkov dostupných na vykonávanie úloh.

- Návrh viacvláknových aplikácií bežiacich na viacprocesorovom systéme je riadený odlišnými predpokladmi než návrh viacvláknových aplikácií pre jednoprocesorový systém. Na jednoprocesorovom systéme si softvérový vývojár môže pomocou niektorých predpokladov zjednodušiť písanie a ladenie programu. Tieto predpoklady však nemusia byť správne na viacprocesorovom systéme. Dve najznámejšie oblasti demonštrujúce tieto odlišnosti sú práca s vyrovnávacou pamäťou a priority vykonávania vlákien. Každý procesor (prípadne jadro procesoru) môže mať svoju vlastnú vyrovnávaciu pamäť. Dáta uchovávané v týchto vyrovnávacích pamätiach môžu byť nesynchronizované. Príkladným problémom môže byť situácia, keď dva rôzne procesory zapisujú do dátových štruktúr v pamäti uložených veľmi blízko seba. Aj keď sú na sebe nezávislé, zápisom jedného z procesorov sa pre druhý procesor označia dáta za neplatné a musia byť znova načítané. Tento problém sa nazýva *falošné zdieľanie*. Na jednoprocesorovom systéme k nemu nemôže dôjsť pretože vyrovnávacia pamäť je zdieľaná medzi všetkými vláknami.

  Priority vykonávania vlákien môžu tiež spôsobiť rozdielnosť chovania sa aplikácie na viacprocesorovom systéme. Predpokladajme, že za cieľom zvýšenia výkonu programátor predpokladá, že vlákno s vyššou prioritou bude bez prerušenia vykonávané na úkor vlákna s nižšou prioritou. Na viacprocesorovom systéme však plánovač operačného systému pridelí vlákno s nižšou prioritou druhému procesoru a obe tak môžu bežať zároveň. Ak programátor optimalizáciu staval na predpoklade, že vlákno s vyššou prioritou bude vykonávané neprerušene, môže dôjsť k nestabilite aplikácie na viacprocesorovom systéme.

2.3 Amdahlov zákon

V tomto bode vyvstáva otázka, ako určiť prírastok výkonnosti aplikácie, keď sa podarí úlohu rozložiť medzi niekoľko nezávislých a súčasne riešiteľných úloh. Intuícia našepkáva, že prírastok by mal byť značný. To však platí len v prípade, že úlohu sa skutočne podarilo celú rozložiť na niekoľko úplne nezávislých častí. Najčastejšie to ale nie je také jednoduché. Ako teda zistiť, aký je skutočný prírastok výkonu aplikácie. Jedným z prístupov je podeliť čas spotrebovaný na vykonanie sekvenčnej verzie algoritmu časom spotrebovaným paralelnou verziou tohoto algoritmu. Výsledok sa nazýva *zrýchlenie*. (S - zrýchlenie, T_s - čas

vykonávania sekvenčného algoritmu, T_p - čas vykonávania paralelného algoritmu, n_t - počet procesorov (vlákien))

$$S(n_t) = \frac{T_s}{T_p}$$

Na základe predchádzajúceho definovania si zrýchlenia je možné určiť limitu prínosu zrýchľovania zo zvyšovania počtu procesorov (a teda aj vlákien). Na tento účel slúži Amdahlov zákon [1] [7], ktorý nám vyjadruje maximálny teoretický prínos paralelizácie úlohy vzhľadom k najlepšiemu možnému sekvenčnému riešeniu úlohy. (S - zrýchlenie, f - pomer paralelizovateľnej časti úlohy, n - počet procesorov (vlákien))

$$S = \frac{1}{(1-f) + (f/n)}$$

Amdahl, autor tohto zákona, vychádzal z jednoduchého úsudku, že celkové zrýchlenie aplikácie získame ako podiel celkového času vykonávania aplikácie a súčtu času sekvenčne vykonávanej časti a paralelizovaním urýchlenej časti.

Amdahlov zákon predpokladá, že paralelizovateľná časť úlohy je dokonale škálovateľná. Vypovedá nám o dosiahnuteľnom zrýchlení pri neobmedzenom zvyšovaní počtu procesorov, pričom najobmedzujúcejším faktorom je z tohto pohľadu sekvenčná časť algoritmu. Napríklad, pri 10% neparalelizovateľnej časti kódu je dosiahnuteľné maximálne 10-násobné zrýchlenie. Je dôležité si uvedomiť, že zvyšovanie počtu procesorov má vplyv iba na paralelizovateľnú časť úlohy, a preto ak je paralelizovateľná iba desatina úlohy, najkratší čas vykonávania úlohy bude vždy najmenej 90% času potrebného na vykonanie sekvenčnej verzie úlohy.

Ďalším záverom vyplývajúcim z Amdahlovho zákona je, že dôležitejšie je zmenšovať sekvenčnú časť úlohy, než zvyšovať počet procesorov riešiacich úlohu. Až keď je úloha z väčšej časti paralelizovaná je prospešnejšie pridávanie ďalších procesorov na riešenie úlohy než ďalšia paralelizácia úlohy.

Pre lepšiu výpovednú hodnotu pre viacvláknové aplikácie môžeme Amdahlov zákon rozšíriť o ďalšiu premennú, a to réžiu spravovania vlákien. (S - zrýchlenie, f - pomer paralelizovateľnej časti úlohy, n - počet procesorov (vlákien), $H(n)$ - réžia spravovania vlákien)

$$S = \frac{1}{(1-f) + (f/n) + H(n)}$$

Táto réžia pozostáva z dvoch hlavných zložiek, a to réžia operačného systému a medzivláknové aktivity ako komunikácia alebo synchronizácia. Z tohto zápisu je zjavné, že réžiu je potrebné držať na minimálnej hodnote. Veľmi zle napísané viacvláknové aplikácie môžu dokonca dosiahnuť zrýchlenie menšie než 1 vďaka tomu, že réžia spravovania vlákien bude neúmerne vysoká. Niekedy môže významne znížiť prínos celej paralelizácie úlohy, preto je dôležité pri paralelnom programovaní brať do úvahy aj tento rozmer.

Kapitola 3

Paralelné programovanie

Správne implementované vlákna môžu podstatne vylepšiť výkon aplikácie. Na druhej strane však nesprávne použitie vlákien môže viesť k horšiemu výkonu, nepredvídateľnému správaniu a ťažko riešiteľným problémom. Dobré pochopenie fungovania vlákien je základom pre prevenciu pred týmito problémami a pre maximálne využitie výkonového potenciálu ukrytého v paralelizme. K tomu, aby sme dokázali naplno využiť prostriedky moderných viacprocesorových a viacjadrových systémov je potrebné pochopiť softvérové vlákna. Na prvý pohľad sa môže zdať práca s vláknami zložitá, po pochopení pár hlavných princípov však každému príde ich podstata jednoduchá.

3.1 Definícia vlákna

Vlákno je postupnosť súvisiacich inštrukcií, ktorá je vykonávaná nezávisle na iných postupnostiach inštrukcií. Každý program obsahuje aspoň jedno vlákno, tzv. hlavné, ktoré inicializuje program začne s vykonávaním inštrukcií. Toto vlákno môže vytvoriť ďalšie vlákna, ktoré pracujú na inej úlohe, alebo nemusí vytvoriť žiadne ďalšie vlákno a môže celú prácu vykonať samo. Každé vlákno má svoj vlastný stav.

Hardvérové vlákno je cesta vykonávania inštrukcií nezávislá na iných cestách vykonávania inštrukcií. Operačný systém mapuje softvérové vlákna na tieto cesty. Voľba paralelizácie aplikácie odráža potreby úlohy a dostupnosť prostriedkov na paralelné vykonávanie úloh. Prílišný paralelizmus môže viesť k zhoršeniu výkonu. Správna miera sa dá nájsť rozumným návrhom a testovaním.

3.2 Typy vlákien

Výpočtový model vlákien pozostáva z troch úrovní:

- *Používateľské vlákna.* Vlákna vytvorené a spravované v rámci aplikácie.
- *Vlákna na úrovni jadra OS.* Spôsob akým väčšina operačných systémov implementuje vlákna.
- *Hardvérové vlákna.* Spôsob akým sa vlákno vykonáva na hardvérových prostriedkoch.

Bežný jednovláknový program často zahŕňa všetky tri úrovne: programové vlákno je implementované operačným systémom ako vlákno na úrovni jadra a vykonané ako hardvérové

vlákno. Medzi týmito vrstvami sú rozhrania, ktoré sú často obsluhované automaticky. Napriek tomu je pre efektívne využitie prostriedkov dobré poznať ako tieto rozhrania pracujú. Nasledujúce odstavce sú venované tejto problematike.

3.3 Vlákna nad úrovňou OS

Najľahšou cestou k vyhnutiu sa problémom s vláknami používanými v aplikácii je uvedomenie si dejov, ktoré vstupujú do hry pri ich použití. V aplikáciách, ktoré nepoužívajú behové prostredie, vytvorenie vlákna znamená volanie systémovej funkcie rozhrania operačného systému. Toto volanie je neskôr za behu vykonané ako volanie funkcie jadra operačného systému pre vytvorenie vlákna. Inštrukcie vlákna sú nasledovne predané procesoru na vykonanie. Vo fáze Definovania a Prípravy sú vlákna špecifikované programovacím prostredím a dekódované prekladačom. Počas Operačnej fázy je vlákno vytvorené a spravované operačným systémom. Nakoniec, vo fáze Vykonávania, procesor vykoná postupnosť inštrukcií vlákna.

Kód aplikácie môže byť závislý na behovom prostredí, v tom prípade sa kód nazýva *riadený kód* a beží v prostredí, ktoré vykonáva aplikačné funkcie a volania operačného systému. Medzi riadené prostredia patria aj Java Virtual Machine (JVM) a Microsoft Common Language Runtime (CLR). Tieto prostredia nevykonávajú žiadne plánovanie spúšťania vlákien. Túto funkciu prenechávajú operačnému systému.

Vo všeobecnosti môžu byť aplikačné vlákna implementované na úrovni aplikácie použitím ustálených API. Najpoužívanejšími sú OpenMP a na nižšej úrovni sú to knižnice ako Pthreads alebo Windows threads. Výber API záleží na potrebách programátora, pričom OpenMP ponúka jednoduchší spôsob práce s vláknami, kým spomenuté knižnice sú vhodné pre potrebu detailnejšieho riadenia práce s vláknami.

3.4 Vlákna na úrovni OS

K pochopeniu vlákien z pohľadu operačného systému je dôležité si uvedomiť, že operačný systémy sa skladajú z dvoch rozdielnych častí: používateľská časť (kde bežia používateľské aplikácie) a jadrová časť (kde sa vykonávajú systémové záležitosti). Medzi týmito dvomi vrstvami je rozhranie, ktoré tvoria systémové knižnice. Tie obsahujú komponenty operačného systému schopné bežať na používateľskej úrovni. Ešte nižšou vrstvou je hardvérová abstraktná vrstva, ktorá tvorí rozhranie medzi operačným systémom a procesorom.

Jadro je základom operačného systému. Spravuje tabuľky, v ktorých uchováva informácie o procesoch a vláknach. Väčšina aktivít spojených s vláknami sa deje práve na tejto úrovni. Knižnice na prácu s vláknami ako OpenMP alebo Pthreads (POSIX vlákna) používajú vlákna na úrovni jadra OS. Windows podporuje oba typy vlákien, používateľské a aj na úrovni jadra. Používateľské vlákna vo Windowse vyžadujú vytvorenie celého systému spravovania vlákien a manuálne plánovanie spúšťania vlákna. Ich výhodou je, že vývojár môže ovplyvniť každý jeden detail v spravovaní vlákna. Vo všeobecnosti však nie sú priveľkým prínosom a je možné sa zaobísť aj bez nich. Vlákna na úrovni jadra ponúkajú vyšší výkon a viaceré vlákna toho istého procesu môžu byť vykonávané na rôznych procesoroch. Réžia spojená s používateľskými vláknami je však menšia a vlákna na úrovni jadra sú často znovu použité po dokončení ich pôvodnej práce.

Procesy sú samostatné programy, ktoré majú svoj vlastný adresný priestor. Sú najzákladnejšou vykonateľnou jednotkou spravovanou ako nezávislá entita vo vnútri operačného

systému. Existuje priama súvislosť medzi procesmi a vláknami. Viaceré vlákna môžu byť obsiahnuté v jednom procese. Všetky vlákna v procese zdieľajú jeden adresný priestor, takže ho môžu využívať na jednoduchú medzivláknovú komunikáciu.

Program môže mať jeden a viac procesov. Jeden proces môže obsahovať jedno alebo viac vlákien. Každé vlákno je mapované na procesor. Koncept *blízkosti procesora* (z angl. *processor affinity*) dovoľuje programátorovi definovať procesor, na ktorom sa bude vlákno prednostne vykonávať. Väčšina operačných systémov podporuje túto vlastnosť, ale vykonávanie na konkrétnom procesore nie je zaručené.

Existuje niekoľko možností mapovania vlákien na procesory: jedna k jednej (1:1), veľa na jeden (M:1) a veľa na veľa (M:N). Model 1:1 nepotrebuje plánovanie vykonávania vlákien podporený v knižnici pre prácu s vláknami. Túto úlohu preberá operačný systém a ide o tzv. *preemptívne vykonávanie viac vlákien* Moderné operačné systémy používajú tento model vlákien. V prípade modelu M:1 knižnica pre prácu s vláknami plánuje spúšťanie vlákien a riadi ich priority. Ide o tzv. *kooperatívne vykonávanie viac vlákien.* Pre model M:N je mapovanie ľubovoľné.

3.5 Vlákna na úrovni hardvéru

Hadrvér vykonáva inštrukcie zo softvérových vrstiev. Inštrukcie aplikácie sú mapované na hardvérové prostriedky a postupne prenikajú cez nižšie vrstvy - operačný systém, behové prostredie - až na hardvér.

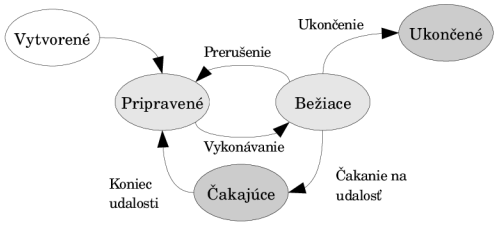
V minulosti bolo na súčasné vykonávanie vlákien potrebných viac procesorov. Dnes je možné spúšťať viaceré vlákna v rámci jedného procesoru na viacerých výkonných jadrách alebo na jednom zdieľanom výkonnom jadre pre dve vlákna. Tu je treba rozlišovať medzi *súbežným* (Hyper-Threading) a paralelným vykonávaním vlákien.

3.6 Vytváranie vlákien

Jeden proces teda môže obsahovať niekoľko vlákien, ktoré pracujú nezávisle jeden na druhom. Napriek tomu zdieľajú jeden adresný priestor a niektoré prostriedky, napr. popisovače súborov. Naviac každé vlákno má svoj vlastný zásobník. Tieto zásobníky sú vo väčšine prípadov spravované operačným systémom, takže vývojár sa nemusí starať o ich veľkosť a alokáciu. Na druhej strane je dobré poznať obmedzenia z toho plynúce pri práci s viacerými vláknami. Veľkosti zásobníkov vlákien sa môžu meniť od systému k systému. Vytváranie veľkého počtu vlákien preto môže znamenať dramatické zníženie výkonu aplikácie.

Po vytvorení sa vlákno vždy nachádza v jednom zo stavov (obr. 3.1): pripravené, bežiace, čakajúce (blokované) alebo ukončené. Existujú aj ďalšie podstavy, ktoré odrážajú dôvody vstupu vlákna do toho-ktorého stavu. Tieto podstavy môžy byť užitočné pri ladení a analyzovaní vlákien aplikácie.

Každý proces obsahuje aspoň jedno vlákno, tzv. *inicializačné vlákno*, ktoré je vytvorené počas inicializácie procesu. Aplikačné vlákna sa vytvárajú počas behu prvotného vlákna. Po vyvorení sa nové vlákno dostane do stavu pripravené. Následne sa pri pokuse vykonávať inštrukcie dostane buď do stavu bežiace alebo blokované. Ak je blokované, čaká na uvoľnenie prostriedkov alebo na iné vlákno. Po dokončení práce sa vlákno dostane do stavu ukončené alebo naspäť do stavu pripravené. Počas ukončovania programu sa ukončia aj všetky jeho vlákna.



Obrázek 3.1: Stavový diagram vlákna

3.7 Výzvy paralelného programovania

Použitie vlákien umožňuje výrazne zvýšiť výkon aplikácie pomocou súčasného vykonávania úloh. Na druhej strane musí vývojár čeliť vyššiemu stupňu zložitosti, ktorý vyžaduje pozorný prístup k riešeniu riadenia tokov v aplikácii. Táto zložitosť plynie z prostého faktu, že v programe dochádza súčasne k viacerým aktivitám. Riadenie súčasných udalostí a ich interakcie vedie k riešeniu nasledujúcich problémov:

- *Synchronizácia* je proces zladenia aktivít dvoch alebo viac vlákien. Napríklad jedno vlákno čaká na dokončenie úlohy v druhom vlákne.
- *Komunikácia* odpovedá za priepustnosť a čas odozvy pri vymieňaní si dát medzi vláknami.
- *Vyváženie výkonu* odpovedá za rovnomerné rozloženie práce medzi viaceré vlákna tak, aby boli zhruba rovnomerne zaťažené.
- *Škálovateľnosť* je schopnosť využitia zvýšeného množstva prostriedkov. Napríklad aplikácia riešiaca úlohu na štvorici procesorov bude schopná zvýšiť výkon riešenia úlohy na ôsmich procesoroch.

Všetky tieto problémy je potrebné starostlivo ošetriť k tomu, aby sa dosiahol čo najvyšší výkon aplikácie. Nasledujúce odstavce ponúkajú konštrukcie pre zvládnutie týchto problémov.

3.8 Synchronizačné prvky

Synchronizácia je mechanizmus na dosiahnutie správneho poradia vykonávania úloh vo vláknach. Riadi vzájomné poradie vykonávania vlákien a rieši tak problémy nežiadaného správania sa. V prostredí, kde sa komunikácia odohráva medzi odosielateľom a prijímateľom je synchronizácia implicitná, keďže prijať správu je možné až po jej odoslaní. V prostredí zdieľanej pamäte je však situácia odlišná, kedže vnútorné závislosti medzi vláknami nie sú jednoznačné. Dajú sa však dosiahnuť pomocou podmienok.

Dva najpoužívanejšie typy synchronizačných mechanizmov sú: vzájomné vylúčenie a synchronizácia podmienkou. V prípade *vzájomného vylúčenia* blokuje jedno vlákno prístup do kritickej sekcie - časti kódu, ktorý obsahuje zdieľané dáta - a ostatné vlákna musia čakať, ak sa chcú dostať do tejto časti kódu. Tento mechanizmus sa používa, ak dve alebo viac vlákien zdieľa pamäťový priestor. Vzájomné vylúčenie je riadené plánovačom a je závislé na

jeho granularite. *Synchronizácia podmienkou* na druhej strane blokuje vykonávanie vlákna, kým nie je splnená nejaká špecifická podmienka.

Existuje množstvo synchronizačných techník. Záleží na programovacom prostredí, ktoré z týchto techník implementuje a ponúka programátorom.

Kritická sekcia je blok kódu, v ktorom sa pristupuje ku zdieľaným premenným, ktoré využívajú rôzne vlákna. Bezpečnosť kritickej sekcie sa dosahuje použitím rôznych synchronizačných prvkov. Ich správnym použitím sa dosiahne, že do kritickej sekcie vstúpi naraz vždy len jedno vlákno. Iným pomenovaním kritických sekcií môže byť *synchronizačné bloky.* V závislosti na spôsobe použitia kritických sekcií je dôležitá aj ich veľkosť. Obecne platí zásada, že velkosť kritických sekcií by mala byť čo najmenšia. Veľké bloky kritických sekcií by mali byť rozdelené na menšie.

Synchronizácia je typicky realizovaná pomocou troch typov prvkov: semafórmi, zámkami a stavovými premennými. Použitie týchto prvkov závisí od požiadaviek aplikácie. Sú implementované atomickými operáciami a používajú príslušné pamäťové zábrany. *Pamäťová zábrana* alebo pamäťová bariéra je operácia závislá na architektúre procesoru, ktorá zabezpečí správne poradie vykonania pamäťových operácií jedného vlákna z pohľadu druhého vlákna. Tieto operácie sú obyčajne ukryté za synchronizačné prvky vyššej úrovne abstrakcie a programátor sa nimi nemusí zaoberať.

Semafóry

Semafóry, prvá skupina prvkov zabezpečujúcich vzájomné vylúčenie v procese paralelnej synchronizácie, boli predstavené známim matematikom Edgarom Dijkstrom. Dijkstra ukázal, že synchronizácia je dosiahnuteľná použitím klasických strojových inštrukcií a hierarchických štruktúr. Navrhol riešenie, v ktorom semafór reprezentuje celé číslo a je ohraničený dvomi atomickými operáciami, P - testovanie a V - inkrementácia. Aj keď tento návrh semafórov prešiel evolúciou, princíp zostavá rovnaký. P predstavuje čakanie na uvoľnenie zdrojov a V predstavuje odstránenie bariéry. Hodnota semafóru je inicializovaná na 0 alebo 1 ešte pred spustením paralelnej úlohy. Operácia P dekrementuje hodnotu semafóru a testuje, či je menšia než nula. Ak áno, vlákno sa zaradí do zoznamu čakajúcich na uvoľnenie zdroja. Operácia V naopak inkrementuje hodnotu semafóru, a ak je menšia alebo rovná nule, uvoľní jedno vlákno zo zoznamu čakajúcich. Operácia P blokuje vlákno, kým hodnota semafóru nedosiahne 0. Operácia V nezávisle na P signalizuje povolenie pokračovania vo vykonávaní. Tieto operácie sú „nedeliteľné“ a vykonávané súčasne. Kladná hodnota semafóru reprezentuje počet vlákien, ktorým je povolené pokračovať v práci bez blokovania. Ak je hodnota rovná nule, žiadne vlákno nečaká. Ak ďalšie vlákno potrebuje dekrementovať hodnotu semafóru, bude zablokované a odloží sa do zoznamu čakajúcich vlákien. Ak je hodnota obmedzená na 0 a 1, semafór sa nazýva *binárny semafór.*

Na semafór môžeme hľadieť ako na čítač s dvoma atomickými operáciami. Implementácie semafórov sa rôznia. Z pohľadu použiteľnosti existujú dve kategórie semafórov: silné a slabé. Silné semafóry dodržiavajú model First-Come-First-Serve, ktorý zaručuje, že každé vlákno príde na rad. Slabé semafóry toto nezaručujú a môže sa stať, že niektoré vlákno sa nikdy alebo veľmi dlhú dobu nedostane na rad.

Semafóry sú však skôr historickou záležitosťou podobne ako neštruktúrovné „goto“. Väčšina programovacích prostredí ponúka štruktúrované synchronizačné prvky na vyššej úrovni abstrakcie. Podobne však ako „goto“ môže byť aj semafór najlepším dostupným riešením. Napríklad v situácii, keď je dostupných niekoľko inštancií zdieľaných prostriedkov, ku ktorým je možné povoliť prístup viac než jedného vlákna. Každá operácia P rezervuje

jednu inštanciu a operácia V uvoľní jednu inštanciu.

Zámky

Zámky sú podobné semafórom s tým rozdieľom, že jedno vlákno ovláda zamknutie jednej inštancie. Zámky ponúkajú dve základné operácie:

- Atomické čakanie na uvoľnenie zámku a následné zamknutie.
- Atomická zmena stavu zámku zo zamknutého na odomknutý.

V každom prípade najviac jedno vlákno zamkýňa zámok. Vlákno musí najprv zamknúť zámok aby mohlo pristupovať k zdieľanému zdroju, inak čaká. Ak vlákno chce získať prístup ku zdieľaným dátam, musí najprv zamknúť zámok, vykonať zmenu dát a následne odomknúť zámok. Stupeň granularity zámku môže byť hrubý alebo jemný. Hrubá granularita zámku znamená vyššiu konfliktnosť. Pre odstránenie problémov granularity väčšina procesorov podporuje operáciu Compare-And-Swap (CAS), ktorá umozňuje implementáciu synchronizácie bez zámku. Atomická CAS operácia zaručuje, že zdieľané dáta zostanú synchronizované medzi vláknami. Ak je použitie zámku nevyhnutné, je vhodné ho vložiť do kritickej sekcie. Z implementačného hľadiska je vždy vhodné používať explicitné zámky definované vývojárom. Obecne však nie je odporúčané držať zámky príliš dlho uzamknuté. Implicitné zámky môžu byť obsluhované aplikačným rámcom (napr. databázovým systémom).

Aplikácia môže používať ľubovoľný druh zámkov v závislosti od požiadaviek úlohy. Nie je však dobré miešať typy zámkov pri plnení jednej úlohy. Dostupné sú tieto druhy zámkov:

Mutexy. Mutexy sú najjednoduchšie použiteľné zámky. Často sú používané na vysvetlenie princípov zámkov. Uvoľnenie mutexu nemusí závisieť iba na vykonaní príslušnej operácie. Môže byť časovo podmienené, takže po vypršaní sa mutex samovoľne odomkne. Použitím časovača alebo try-finally klauzule sa dá predísť uviaznutiu.

Rekurzívne zámky. Rekurzívne zámky môžu byť zamknuté opakovane viac než jeden krát vláknom, ktoré ho zamklo prvý raz. Uvoľnenie zámku nastane až vtedy, keď ho vlastniace vlákno odomkne raz pre každé zamknutie. Tento typ zámkov je vhodný pre rekurzívne funkcie. Je však potrebné si dávať pozor na počet zamknutí a odomknutí zámku. Rekurzívne zámky sú pomalšie než nerekurzívne.

Read-Write zámky. Read-Write zámky povoľujú súčasné čítanie viacerých vlákien, ale zápis je obmedzený iba na jedno vlákno. Tento typ zámkov je použiteľný v prípade, že niekoľko vlákien súčasne číta zdieľané dáta, ale zapisuje iba jedno. Veľké objemy dát je výhodnejšie rozdeliť na menšie úseky, na ktorých má prístup vždy iné vlákno, než držať zámok po dlšiu dobu.

Spin zámky. Spin zámky sú neblokujúce zámky vlastnené vláknom. Čakajúce vlákna musia dokola čakať na uvoľnenie zdieľaného prostriedku. Tento druh zámkou sa používa výlučne na multiprocesorových systémoch, pretože vlákno čakajúce na uvoľnenie zámku spotrebúva všetok procesorový čas. Výhodou použitia tohto typu zámku je v prípade, že nebude dlho zamknutý, to, že vlákno nebude zdržiavané prepínaním na iné vlákno. Prepnutie totiž vyžaduje prepnutie kontextu a aktualizáciu štruktúr vlákna, čo môže trvať dlhšie než odomknutie spin zámku. Problémom však môže byť vyhladovenie ostatných vlákien,

ak je nesprávne použité. Neodporúča sa napr. držať zamknutý spin zámok počas volania funkcie nejakého podsystému.

Stavové premenné

Stavové premenné sú tiež založené na princípe Dijkstrových semafórov s tým rozdielom, že operácie týchto synchronizačných prvkov nemajú asociovanú žiadnu uloženú hodnotu. To znamená, že stavové premenné neobsahujú testovanú hodnotu, až stav zdieľaných dát je použitý na rozhodnutie podmienky. Vlákno čaká alebo povoľuje vykonávanie spolupracujúcich vlákien na základe splnenia podmienky. Stavové premenné sú používané v prípade, keď je potrebný istý stupeň plánovania vykonávania vlákien. Atomické operácie stavových premenných sú nasledovné (stavová premenná C, využívajúca zámok L):

- wait(L): Atomické odomknutie zámku a čakanie, opätovne uzamkne zámok.
- signal(L): Povolí beh jedného z čakajúcich vlákien, opätovne uzamkne zámok.
- broadcast(L): Povolí beh všetkých čakajúcich vlákien, opätovne uzamkne zámok.

Tento mechanizmus sa dá s výhodou použiť na riadenie skupiny vlákien pomocou signálov. Rozhlásenie môže byť náročná operácia na výkon, preto je potrebné uvážiť jej použitie. V niektorých prípadoch ale môže byť efektívnym riešením.

3.9 Správy

Správa je špecifický spôsob komunikácie na prenos informácie alebo signálu z jednej domény do druhej. Definícia domény je rôzna pre rôzne scenáre. Pre viacvláknové prostredie je doménou vlákno. Koncept správ je definovaný skôr v súvislosti s procesmi. Z pohľadu výmeny správ rozlišujeme vnútroprocesovú, medziprocesovú a proces-proces komunikáciu. Dve vlákna patriace tomu istému proceu môžu využiť vnútroprocesovú komunikáciu. Dve vlákna patriace dvom rôznym procesom môžu využiť medziprocesovú komunikáciu. Najčastejšou formou komunikácie pomocou správ z pohľadu programátora je komunikácia dvoch procesov.

Vo všeobecnosti, správy môžu byť delené podľa pamäťového modelu prostredia, v ktorom sú vymieňané. Posielanie správ v zdieľanom pamäťovom modeli musí byť synchrónne, naopak v distribuovanom pamäťovom modeli asynchrónne. Z iného pohľadu môže byť delenie na synchrónne a asynchrónne správy závislé od toho, či je potrebné po odoslaní správy čakať na prijímateľa alebo sa môže pokračovať vo vykonávaní úlohy.

Na synchronizáciu vlákien sa používajú semafóry, zámky a stavové premenné. Tieto synchronizačné prvky sprostredkúvajú informácie o stave a prístupových právach. Na prenos dát je použitá *medzivláknová komunikácia*. V prípade medzivláknovej komunikácie zostáva synchronizácia explicitná, keďže po obdržaní správy sa potvrdzuje jej doručenie. Potvrdenie odstraňuje problémy s uviaznutím a dátovými konfliktmi. Na úrovni hadrvéru môže komunikácia preberať formu výmeny hodnôt medzi registrami, dátami vo vyrovnávacej pamäti alebo hlavnej pamäti.

3.10 Koncepty toku riadenia

Na synchronizáciu akcií v paralelnej aplikácii sa používajú mechanizmy, medzi ktoré patria aj zábrana (z angl. fence) a beriéra. Nasledujúce odstavce predstavujú tieto dva koncepty.

Zábrana

Mechanizmus zábrany je implementovaný pomocou inštrukcie, takže sa často hovorí aj o inštrukcii zábrany. Na viacprocesorových systémoch so zdieľanou pamäťou zaručuje zábrana pamäťovú konzistentnosť operácií. V priebehu vykonávania tejto inštrukcie sa zaručuje dobehnutie všetkých predošlých pamäťových operácií a odloženie všetkých nasledujúcich pamäťových operácií, až kým sa nedokončí inštrukcia zábrany. Zabezpečuje správne mapovanie softvérového pamäťového modelu na hardvérový. Explicitné použitie inštrukcie zábrany konkrétnej platformy sa však neodporúča. Lepšie je spoľahnúť sa na prostriedky prekladačov.

Bariéra

Bariéra je mechanizmus na synchronizáciu skupiny vlákien spolupracujúcich na spoločnej úlohe, až kým nedosiahnu spoločný bod v toku riadenia. Pomocou tejto metódy je možné dosiahnuť stav, keď vykonanie ďalšej úlohy začne až vo chvíli, keď všetky ostatné vlákna dokončili svoju predošlú úlohu. Zaručuje, že žiadne vlákno nebude pokračovať v práci, kým sa všetky ostatné nedostanú do spoločného logického bodu vo vykonávaní úlohy.

3.11 Najčastejšie problémy

Paralelné programovanie je známe už desaťročia, nevzniklo uvedením viacjadrových procesorov, bolo však skôr výnimočným javom. Programátori sa tak už mali možnosť stretnúť s najčastejšími problémami, na ktoré vznikli účinné riešenia. Poznanie týchto riešení počas návrhu viacvláknovej aplikácie umožňuje predchádzať chybám, ktoré nie je možné jednoduchou opravou odstrániť v pokročilejších fázach projektu. Je potrebné s nimi rátať už od začiatku. Nasledujúce odstavce prinášajú prehľad týchto najčastejších problémov, ich symptómy a postupy, ktorými je možné im predchádzať.

Priveľa vlákien

Keď niekoľko vlákien vykoná prácu rýchlejšie ako jedno vlákno, viac vlákien musí tú istú prácu vykonať ešte rýchlejšie. V skutočnosti však použitie privysokého počtu vlákien vyústi do priveľkej réžie prepínania kontextu vlákien, ktoré v konečnom dôsledku môže z výkonu skôr ubrať než pridať. Ďalším problémom je, že použitie väčšieho počtu paralelných vlákien než je možné namapovať na hardvérové prostriedky, so sebou prináša nutnosť tieto prostriedky zdieľať.

Pri väčšom počte softvérových vlákien než je dostupných hardvérových, musí operačný systém naplánovať beh niekoľkých softvérových vlákien na jednom hardvérovom. Procesorový čas je tak pridelovaný viacerým vláknam po tzv. časových kvantách, po uplynutí ktorého je vlákno prevedené do stavu pozastavenia a iné vlákno sa dostáva do stavu vykonávania. Softvérové vlákno je teda zastavené a nevykonáva užitočnú prácu. Tento mechanizmus zaisťuje, že všetky vlákna dosiahnu pokrok vo vykonávanej úlohe a nijaké nebude na dlhší čas blokované vykonávaním ostatných. Každé prepnutie vlákna však znamená spotrebovaný procesorový čas, ktorý ide na vrub užitočnej práci. Dôsledkom toho je, že pri použití väčšieho počtu vlákien dosiahneme nižší výkon systému pri výpočte úlohy než keby sme použili menší počet softvérových vlákien.

Na prvý pohľad je réžia prepínania vlákien spôsobená najmä uchovávaním stavu registrov hardvérového vlákna. Pozastavenie vlákna vyžaduje uloženie stavu registrov tak, aby

pri zobudení mohol byť stav opäť navodený. Plánovač operačného systému však prideľuje vláknam dostatočne veľké časové kvantá na to, aby bol čas potrebný na tento úkon zanedbateľný. Oveľa väčším problémom je stav vyrovnávacej pamäte vlákna. Vyrovnávacie pamäte moderných procesorov sú 10 až 100 násobne rýchlejšie než hlavná pamäť. Takisto je pri práci s vyrovnávacou pamäťou vylúčená komunikácia po systémovej zbernici. Vyrovnávacia pamäť však býva obyčajne podstatne menšia než hlavná pamäť a po zaplnení je nutné z nej dáta odstraňovať v prospech uvoľnenia pamäte pre nové spracovávané dáta. Ak vlákna pracujú na rôznych častiach úlohy, je veľmi pravdepodobné, že budú pracovať s rozdielnymi dátami. Preto je možné, že po prepnutí kontextu bude práve vykonávané vlákno požadovať iné dáta vo vyrovnávacej pamäti než aké sú dostupné. Začne sa proces vyprázdňovania vyrovnávacej pamäte a načítanie potrebných dát. Po opätovnom prepnutí vlákna nastane opäť rovnaká situácia - vlákno požaduje informácie, ktoré boli z vyrovnávacej pamäte vyhodené. Vlákna si tak môžu navzájom prepisovať dáta vo vyrovnávacej pamäti, čo môže podstatne degradovať výpočtový výkon. Do podobnej situácie sa môžeme dostať aj so systémom virtuálnej pamäte, ktorý odkladá nepoužívané stránky pamäte na pevný disk, ak je hlavná pamäť zaplnená. Ak aplikácia pracuje s dátami, ktoré nie je možné umiestniť naraz do hlavnej pamäte, je tento problém dosť reálny. Navyše každé vlákno má svoj zásobník a svoje dátové štruktúry, ktoré by v extrémnom prípade prehnaného množstva vlákien mohli spôsobiť zaplnenie virtuálnej pamäte procesu.

Riešení problému s priveľkým počtom vlákien je používať vždy primerané množstvo softvérových vlákien, najlepšie rovný počtu hardvérových vlákien tak, aby mohli súčasne bežať bez prerušenia. Úplne najlepším riešením je nechať počet vlákien voľný, čiže nie pevne daný aplikáciou, takže bude možné počet softvérových vlákien určiť podľa možností systému, na ktorom beží program. Na rozdiel od neblokovaných vlákien, pripravených na beh, však existujú aj vlákna, ktoré sú blokované napríklad čakaním na udalosť, akou je načítanie bloku dát z pevného disku alebo stlačenie tlačidla myši. Tieto typy vlákien nedostávajú priradené kvantá času pri prepínaní plánovačom úloh a teda ani nespotrebúvajú výpočtový čas. Je užitočné pri návrhu aplikácie oddeliť tieto dva typy vlákien. Vlákna pracujúce na úlohe tak nie sú prerušované a nemusia čakať na udalosti systému, kým blokované vlákna môžu spracovávať prichádzajúce udalosti a po ich obslúžení sa vrátiť do stavu čakajúci.

Dátové konflikty

Nesynchronizovaný prístup do zdieľanej pamäte môže spôsobiť zápisové konflikty, keď výsledky programu sú nedeterministicky závislé na relatívnom poradí vykonávania vlákien. Dátové konflikty môžu byť skryté aj za zjednodušené zápisy príkazov, napr. aktualizácia premennej `x += 1` je typicky vykonaná ako `temp = x; x = temp + 1;`, kde prerušenie vykonávania môže prísť za prvou inštrukciou. Iné vlákno môže počas tohto prerušenia prepísať premennú `x`, avšak následným návratom do pôvodného vlákna je táto hodnota zahodená. A aj v prípade, že je akcia reprezentovaná jedinou inštrukciou, môže nastať situácia, že v hardvéri je rozdelená na niekoľko prekladaných zápisov a čítaní.

V niektorých prípadoch je využitie dátových konfliktov zámerné. Napríklad, ak vlákna asynchrónne aktualizujú nejakú „poslednú aktuálnu hodnotu“. V takom prípade sa musí dávať pozor na to, aby boli zápis a čítanie atomické operácie. V opačnom prípade môže byť hodnota poškodená prekladaným zápisom. Typicky sa to deje pri hodnotách väčších než dvojbajtových, keď zápis musí byť uskutočnený na viackrát. Napr. 80-bitová hodnota desatinného čísla nemusí byť zapísaná ani čítaná atomicky, samozrejme v závislosti na

architektúre. Podobne ani nezarovnané zápisy a čítania môžu byť neatomické.

Dátové konflikty môžu nastať aj v prípade synchronizovaných dátových operácií, ak sú synchronizované na nedostatočnej úrovni. Napr. pri implementácii abstraktného dátového typu množiny môže nastať situácia, že po testovaní prítomnosti prvku v množine pridáme nový prvok. Ak sa však po prejdení celej množiny prepne kontext vlákna a pridá sa prvok, ktorý sme hľadali, v ďalšom výkonávaní pôvodného vlákna sa opakovane pridá ten istý prvok. Tu je požadované, aby bolo atomickou operáciou prehľadanie celej množiny a pridanie nového prvku.

Z toho, čo bolo vyššie napísané vyplýva, že zamykanie operácií na nižšej úrovni môže byť zbytočné mrhanie času, keď operácie na vyššej úrovni potrebujú zvlášť systém zámkov. V tom prípade môžu nízkoúrovňové zámky spôsobiť sbytočnú stratu výkonu aplikácie. Samozrejme sú prípady, keď je dobré, ak majú komponenty programu svoj vlastný interný systém zámkov. Takým prípadom sú napr. knižnice zabezpečené pre beh na viacerých vláknach.

Uviaznutia

Prístupové konflikty sú často riešené použitím zámkov, ktoré vylúčia chybnú hodnotu spôsobenú prekrytím zápisov a čítaní na pamäťové miesto. Nanešťastie aj zámky prinášajú svoje riziká, najbežnejšie uviaznutia. Typickou situáciou vedúcou k uviaznutiu je alokovanie zámku A jedným vláknom a zámku B druhým vláknom a následnou požiadavkou oboch vlákien na získanie zámku toho druhého vlákna. Ani jedno vlákno sa nevzdá svojho alokovaného zámku a teda obe čakajú na seba navzájom. Aj keď je uviaznutie najčastejšie spájané so zámkami, nastať môže pri pokuse o výlučné pridelenie akýchkoľvek iných dvoch jednotiek zdieľaných prostriedkov. Napr. pri pokuse o získanie exkluzívneho prístupu k súboru.

Uviaznutie môže nastať iba v prípade, ak sú splnené všetky nasledujúce podmienky:

1. Prístup k prostriedku je výlučný.
2. Vláknu je dovolené mať alokovaný prostriedok počas žiadania o ďalší.
3. Žiadne vlákno sa nevzdá prideleného prostriedku.
4. Existuje cyklus závislostí žiadostí o pridelenie prostriedkov, v ktorom je prostriedok pridelený jednému vláknu a požadovaný iným vláknom.

Uviaznutie môže byť odstránené porušením ktorejkoľvek z vyššie uvedených podmienok.

Najlepším spôsobom odstránenia uviaznutia je duplikovanie prostriedku, ktorý vyžaduje výlučný prístup, takže každé vlákno by malo svoju vlastnú kópiu. Tým by získalo prístup k prostriedku bez potreby ho uzamknúť. Na záver by bolo možné spojiť duplikované prostriedky do jedného, ak by bolo treba. Odstránením potreby použitia zámku sa vylúči uviaznutie, keďže nemôže dojsť ku konfliktu medzi vláknami. Ak nie je možné duplikovať prostriedky, najlepšou prevenciou proti uviaznutiu je prideľovanie podľa poradia. To znamená, že ak chce vlákno alokovať zámok B, musí mu byť najprv pridelený zámok A. Príkladné uviaznutie z úvodu podkapitoly by tým pádom nemohlo nastať.

Riešení usporiadania zámkov môže byť niekoľko, závislých od danej situácie. Ak sú zámky pomenované, ich usporiadanie môže byť odvodené od abecedného usporiadania ich názvov. Pre zámky v dátovej štruktúre môže byť usporiadanie závislé od topológie štruktúry. Napr. stromové usporiadanie dát môže viesť k poradiu prideľovania zámkov od vyššie umiestnených prvkov po nižšie umiestnené (vnorené) prvky. Ak však neexistuje

žiadne rozumné usporiadanie, je možné použiť poradie alokovania zámkov podľa pamäťovej adresy. Nato je ale potrebné dopredu poznať všetky zámky, o ktoré je žiadané.

V prípade veľkých softvérových projektov je obvyklé, že každý programátor pracuje na svojej komponente a nie je mu známe vnútorné poradie alokovania zámkov v inej komponente. Preto je dobré, ak počas volania vonkajšej metódy z komponenty nie sú uzamknuté žiadne zámky. V opačnom prípade by mohlo dojsť ku skrytej medzikomponentovej závislosti, ktorá by mohla viesť k uviaznutiu.

Treťou podmienkou uviaznutia je, že vlákna sa nevzdávajú už pridelených prostriedkov. Porušenie tejto podmienky u mutexov umožňuje mechanizmus „try lock“, čiže pokus o alokáciu prostriedku. V prípade, že sa nepodarí získať výlučný prístup k prostriedku, môže vlákno uvoľniť ostatné pridelené prostriedky, ktoré tak už nebudú zbytočne blokované a ostatné vlákna tak môžu byť ušetrené uviaznutia. Tento postup je užitočný aj v prípade, že zoraďovanie zámkov je príliš nepraktické.

Blokovanie

Použitie zámkov na odstránenie prístupových konfliktov môže v istých prípadoch spôsobiť zníženie výkonu aplikácie. Zámky sú ako závory na diaľnici. Ak autá prichádzajú rýchlejšie než je závora schopná autá púšťať, vytvárajú sa kolóny. Podobne je to s vláknami, čakajúcimi na uvoľnenie zámku. Ak sa dožaduje viacero vlákien jedného zámku, všetky musia čakať, kým nie je odbavená výlučná sekcia vykonávaného vlákna a až potom môže pokračovať ďalšie vlákno.

Inverzia priorít. Niektoré implementácie vlákien umožňujú priradenie priorít vláknam. Ak je potom vykonávaných niekoľko softvérových vlákien na jednom hardvérovom vlákne, je prednostne spúšťané vlákno s vyššou prioritou než vlákno s nižšou prioritou. Môže však dôjsť k paradoxným situáciám, keď vlákno s nižšou prioritou blokuje vykonávanie vlákna s vyššou prioritou, ktoré čaká na uvoľnenie zámku. Zámok sa však neuvoľní, pretože vlákno s vyššou prioritou, než to so zámkom, dostáva prioritne všetok pridelovaný čas. Vlákno so stredne vysokou prioritou tak beží prednostne pred vláknom s najvyššou prioritou.

Tento problém býva riešený tzv. *dedením priority*. Vláknu, ktoré má vo vlastníctve zámok, na ktorý čaká vlákno s vyššou prioritou, sa na čas zvýši priorita tak, aby bola kritická sekcia čo najrýchlejšie vykonaná. Po uvoľnení zámku sa priorita vráti na pôvodnú úroveň a vlákno s vyššou prioritou môže pokračovať v práci. Ďalšou možnosťou je určenie hornej hranice priority, ktorej môžu dosiahnu vlákna žiadajúce o mutex a pri pridelení mutexu zvýšiť prioritu vlákna na túto úroveň. Prvá metóda je však lepšia v tom, že zbytočne nezvyšuje prioritu vlákna nad potrebnú úroveň.

Odstránenie blokovania

Problém blokovania vlákna zamknutým zámkom nie je riešiteľný použitím rýchlejších implementácií, i keď zámky sú často pomalé. Nahradením zámku jeho rýchlejšou variantou získame konštantné zrýchlenie aplikácie. Nerieši to problém škálovateľnosti. Riešením môže byť už spomínané duplikovanie prostriedkov, ktoré odstraňuje potrebu použitia zámku. Ak máme napr. v programe počítadlo nejakých udalostí, nie je potrebné zamykať prístup k počítadlu pre každé vlákno. Stačí ak budú vlákna počítať udalosti zvlášť a po vykonaní úlohy sa všetky počítadlá spočítajú.

Ak zamknutie prostriedku nie je možné vynechať, je lepšie prostriedok rozdeliť na časti a zamykať tie. To môže odbremeniť najvyťaženejšie zámky. Napríklad, ak chceme realizovať prístup viacerých vlákien k jednej transformačnej tabuľke, do ktorej sa pokúša vložiť nový prvok niekoľko vlákien. Najjednoduchší spôsob je použiť jeden zámok pre celú tabuľku. Viacero žiadostí o zápis do tabuľky však môže znamenať veľkú výkonovú stratu. Lepším riešením je rozdeliť tabuľku na niekoľko podtabuliek, z ktorých každá má svoj vlastný zámok. Pri pokuse o zápis nového prvku do tabuľky sa zamkne iba príslušná časť, kým ostatné zostanú iným vláknam naďalej prístupné. Pri dostatočnom počte podtabuliek a dobrej transformačnej funkcii sa záťaž rovnomerne rozdelí medzi zámky jednotlivých podtabuliek.

Hlavná myšlienka rozdelenia záťaže medzi viacero zámkou vedie k tzv. *jemnozrnnému zamykaniu*. Napríklad, transformačné tabuľky sú najčastejšie implementované ako polia zoznamov, ktoré obsahujú položky s rovnakou hodnotou transformačnej funkcie ako prvku poľa. Vlákna bežiace paralelne môžu súčasne pristupovať k rôznym zoznamom. Toto je výsledkom prostého riešenia, keď sa veľkosť poľa nemení. Ak umožníme zmenu veľkosti poľa, situácia sa radikálne mení, pretože počas zmeny veľkosti musí byť prístup výlučný iba pre vlákno, ktoré zmenu vykonáva. Dá sa to však vyriešiť použitím zámku typu čítač-zapisovač. Použitie tohto typu zámku je užitočné aj v prípade, že je dátová štruktúra často čítaná a zriedkavo zapisovaná. Zámok typu čítač-zapisovač rozlišuje čítačov a zapisovačov. Zámok môže vlastniť viacero čítačov, avšak iba jeden zapisovač naraz. Čítači nemôžu získať zámok, ak ho vlastní zapisovač a naopak. Čítači teda prichádzajú do konfliktu iba so zapisovačmi. Použitie tohto typu zámku je nasledovné. Pole transformačnej tabuľky je určené popisom, na ktorý získa vlákno, ktoré chce pristupovať k zoznamu, zámok typu čítač. Ďalej musí dostať pridelený zámok k zoznamu. Vlákno požaduje prvý zámok typu čítač z toho dôvodu, že nepotrebuje meniť popis poľa, aj keď chce meniť obsah zoznamu. Ak chce vlákno zmeniť veľkosť transformačnej tabuľky, alokuje zámok typu zapisovač. Po pridelení môže meniť popis poľa bez rizika dátového konfliktu. Výhodou tohto riešenia je, že ak chcú vlákna iba čítať zoznamy tabuľky, môžu tak urobiť simultánne. Nevýhodou je, že pri každom prístupe musí vlákno žiadať o dva zámky, čo môže byť zbytočné najmä v prípade, ak sú súčasné prístupy k tabuľke zriedkavé.

Kapitola 4

Intel Threading Tools

Vývoj viacprocesorových architektúr a podporných nástrojov vo firme Intel prebieha už niekoľko desaťročí. Vývojárom ponúkajú celý sortiment softvéru pre vývoj, ladenie a výkonové dolaďovanie aplikácií. Patrí sem prekladač jazyka C/C++ a Fortran, ladiaci nástroj Intel Debugger, analyzačný nástroj výkonu Intel VTune Performance Analyzer, knižnice matematických funkcií a pod. Táto kapitola sa zameriava na podporné nástroje pre ladenie viacvláknových aplikácií Intel Thread Checker a Intel Thread Profiler. Písanie programov využívajúcich paralelizmus pozostáva z rovnakých krokov ako písanie sekvenčných programov. Sú tu však rozdiely, ktoré sú pre mnohých programátorov nové a je potrebné sa s nimi zoznámiť.

Prvým krokom je vždy zápis programu vo zvolenom programovacom jazyku. Avšak len vo výnimočných prípadoch (ak vôbec) bola pri návrhu jazyka braná do úvahy aj podpora konštrukcií pre prácu s paralelizmom. Preto sa táto podpora rieši v mnohých prípadoch využitím knižnice alebo pridaním direktív pre prekladač jazyka (OpenMP). V ďalšom kroku je potrebné odstrániť vzniknuté chyby, ktoré programátor omylom do svojho kódu zaniesol. Paralelizmus v tomto bode opäť prináša podstatné zmeny, pretože sa rozširuje repertoár možných konfliktov a prehreškov v kóde. Štandardný postup vyhľadania a opravenia chyby je ale možné zameniť za automatizované vyhľadanie konfliktných miest pomocou nástroja Intel Thread Checker. Aj keď chyba nebola detegovaná počas spustenia aplikácie, môže tento nástroj označiť kritické miesta, ktoré sú náchylné na uviaznutie alebo dátový konflikt, vďaka zisteniu nesprávneho zaobchádzania so zdieľanou pamäťou medzi vláknami. Posledným krokom je vyladenie výkonu aplikácie, ktoré je najlepšie vykonávať nástrojmi nezasahujúcimi do kódu programu. Človek vďaka nim získava prehľad o dianí v systéme a nachádza kritické miesta, ktoré vyžadujú vylepšenie. To platí rovnako pre sekvenčné programy ako aj tie využívajúce paralelizmu. Intel VTune Performance Analyzer a Intel Thread Profiler sú schopné priniesť vývojárom presný obraz rozloženia výkonu medzi vláknami a jednotlivými časťami kódu aplikácie.

4.1 Intel Thread Checker

Intel Thread Checker [5] je nástroj na detekciu chýb v interoperabilite vlákien vo viacvláknových aplikáciách. Táto trieda chýb je veľmi zákerná a nie je ľahké ich nájsť a odstrániť, pretože aj na pohľad správne fungujúci program v sebe môže ukrývať vážnu chybu, ktorá sa prejavuje nedeterministicky. Jej prejavy sa môžu meniť od spustenia k spusteniu a pri klasickom ladení ladiacim nástrojom sa nemusí vôbec prejaviť.

Intel Thread Checker vytvára na rôznych miestach programu diagnostické správy, ktoré môžu odhaliť nederministické správanie sa medzi vláknami. Týmto spôsobom identifikuje problémy ako dátové konflikty, uviaznutia, mŕtve vlákna, stratené signály alebo zabudnuté zámky. Podporuje analýzu viacvláknových aplikácií využívajúcich OpenMP, POSIX threads alebo Windows API.

Chyby, typické pri použití vlákien v programe, ako dátový konflikt, ktorý môže spôsobiť, že program dáva odlišné výsledky než jeho sekvenčná verzia, alebo uviaznutie, ktoré môže spôsobiť zacyklenie aplikácie, sa detegujú pomocou Intel Thread Checker nasledujúcim spôsobom. Aplikácia sa spustí a zároveň, počas jej behu, sú zhromažďované diagnostické informácie, ktoré sa po dobehnutí programu analyzujú a vývojárovi je predostretý zoznam chýb a upozornení, ktoré sa počas trasovania našli. Zoznam je usporiadaný podľa vážnosti problému, aby sa vývojár mohol zamerať na odstránenie najvážnejších problémov v kóde. Každý problém je viazaný ku konkrétnemu riadku a je zobrazený kontext, zásobník volaní, príslušné premenné a krátka správa o podstate chyby.

Intel Thread Checker vykonáva analýzu programu na základe vstavanej binárnej inštrumentácie, takže nezáleží na prekladači použitom pre zostavenie programu. To je dôležité z hľadiska použitia dynamicky linkovaných knižníc, pre ktoré sú často zdrojové kódy nedostupné. Za použitia prekladača od Intelu sú však výsledné informácie o čosi bohatšie vďaka prekladačom pridanej inštrumentačnej funkcionalite, napr. možnosti podrobnejšej analýzy premenných.

4.2 Intel Thread Profiler

Intel Thread Profiler [6] je modul pre nástroj Intel VTune Performance Analyzer. Intel VTune Performance Analyzer ponúka vývojárom možnosť merať výkon vytvorenej aplikácie zistením času stráveného v jednotlivých častiach programu a zobraziť graf volaní podprogramov pre jednotlivé procesy a pre vlákna, ktoré obsahujú. Dnešné aplikácie dosahujú vysoký stupeň zložitosti a nájsť v nich problémové miesta môže byť veľmi ťažké. Preto je dôležité mať nástroj, ktorý na problémové miesta poukáže napríklad prostredníctvom výkonových štatistík.

Po analýze rozloženia času stráveného v jednotlivých miestach kódu je možné určiť kritické body, ktoré zaberajú najviac procesorového času. Vývojár týmto spôsobom môže dospieť k záveru, že tieto miesta by bolo dobré optimalizovať alebo paralelizovať, aby sa tým zvýšil celkový výkon aplikácie. Podobný postup platí aj pre ladenie výkonu vlákien pomocou Intel Thread Profiler.

Intel VTune Preformance Analyzer je schopný nájsť moduly, funkcie, vlákna alebo aj riadok kódu, ktorý spotreboval najviac cyklov procesoru. Nie je k tomu potrebný žiaden zásah do binárneho kódu aplikácie. Pre zobrazenie riadkov v zdrojom kóde je potrebné zahrnúť symbolické informácie do binárneho kódu aplikácie.

Ladenie výkonu môže viesť k situáciám, keď sa časti kódu podarí paralelizovať a využiť tým ďalší procesor. Dosiahnutý výkon tak môže vzrásť až dvojnásobne. Tu si však treba uvedomiť obmedzenia, ktoré vyjadruje Amdhalov zákon. Treba mať preto striedme očakávania.

Nástroj ponúka možnosť nahliadnuť na graf volaní podprogramov, ktorý môže odhaliť kritické miesto, pre ktoré by bolo vhodné vytvoriť nové vlákno, v ktorom by prebehlo jeho vykonanie. Ďalej ponúka grafické zobrazenie rozloženia časových vzoriek spotrebovaných jednotlivými modulmi alebo vláknami. Vývojár tak získa predstavu o rovnomernosti rozloženia záťaže.

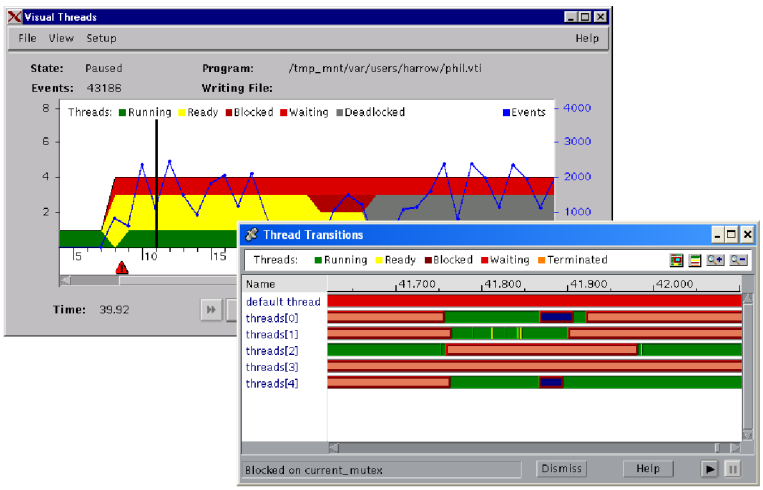
Intel Thread Profiler ako rozšírenie funkcionality Intel VTune Preformance Analyzer využíva poznatky o synchronizačných objektoch medzi vláknami a tým umožňuje podrobnejšie pochopiť príčiny výkonnostných problémov. Dokáže identifikovať prípady, kedy vlákno čaká na dokončenie práce iného vlákna, a kedy iba mrhá procesorovým časom. Využitím informácií o synchronizačných objektoch je schopný zobraziť kritickú cestu, prepínajúcu sa medzi vláknami, ktorej optimalizáciou je vývojár schopný skrátiť čas potrebný na vykonanie užitočnej práce. Intel Thread Profiler dokáže využiť štatistické funkcie knižnice OpenMP a ponúknuť tak lepší prehľad o paralelizovanej časti kódu. Na časovej osi je schopný zobraziť časové príspevky jednotlivých vlákien celkovému spracovaniu úlohy aj mimo kritickej cesty. Programátor tak dostáva do rúk mocný nástroj, ktorý pomáha ladiť výkon priamym zobrazením problémových miest a nie je nútený používať neefektívnu metódu pokus-omyl.

4.3 Alternatívne nástroje

Existuje aj niekoľko ďalších nástrojov na ladenie výkonu paralelných aplikácií. Niektoré nástroje sú určené na ladenie výkonu jednovláknových programov, pričom používajú hardvérové počítadlá, ktoré sú implementované v procesoroch. Týmito prostriedkami je možné zistiť počty výpadkov vo vyrovnávacích pamätiach, počty nesprávne predpovedaných skokov, čas spotrebovaný sekciami kódu a pod. Príkladom takýchto nástrojov sú open source projekty **OProfile** alebo **gprof**, komerčná aplikácia AQtime. Spomenuté programy však ponúkajú len veľmi obmedzené možnosti v profilovaní viacvláknových aplikácií.

HP Visual Threads

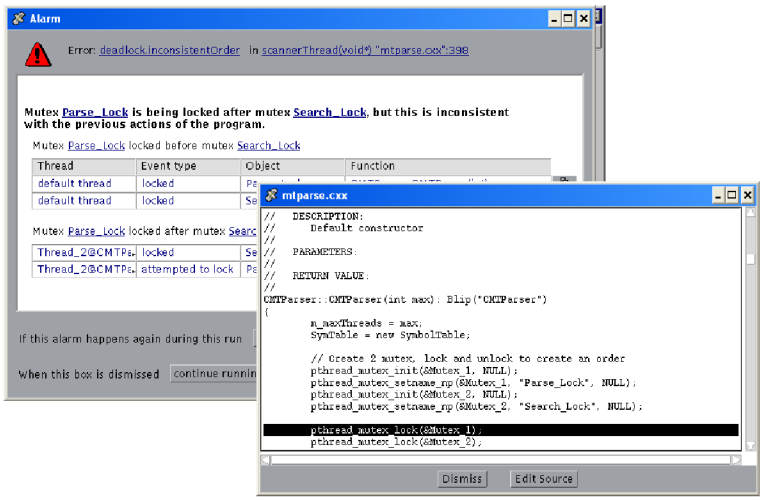
Na ladenie viacvláknových aplikácií vznikol product Visual Threads, ktorý začala vyvíjať firma Compaq. Po akvizícii firmou Hewlett Packard sa s HP Visual Threads zamerali na vývoj aplikácií na platformách komerčných Unixov (Tru64 UNIX, OpenVMS a HP-UX Itanium), predtým bol dostupný aj pre komerčné distribúcie Linuxu – Red Hat a SuSE. Podporuje štandardné POSIX vlákna a umožňuje:



Obrázek 4.1: HP Visual Threads profilovanie vlákien.

Automatickú analýzu

- chýb vzťahujúcich sa k programovaniu s vláknami
- inverzií priorít, uviaznutí, prístupových konfliktov
- kritických miest z hľadiska výkonu



Obrázek 4.2: HP Visual Threads detekcia chýb.

Interaktívnu analýzu

- vyhodnotenie štatistických meraní
- detekciu určitých stavov aplikácie

Zobrazenie

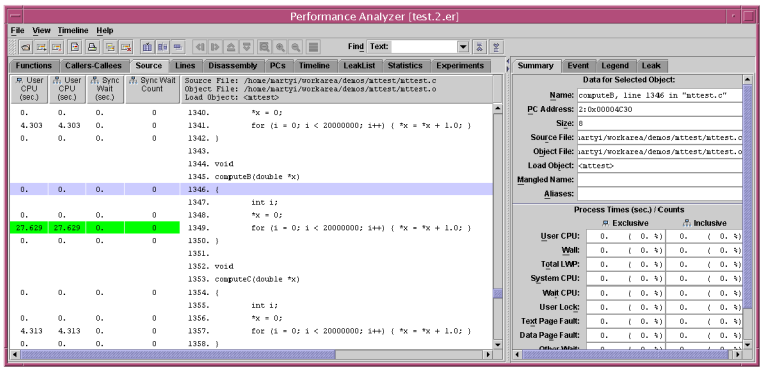
- grafov vykonávania vlákien
- štatistických grafov
- vyťaženia zámkov

Aplikáciu som však nemohol vyskúšať, keďže som nemal k dispozícii systém s nainštalovaným komerčným Unixom. Podľa popisu z dostupných zdrojov [11] to však vyzerá byť mocný nástroj s množstvom užitočných funkcií.

Sun Studio Performance Tools

Z dielne spoločnosti Sun Microsystems je súbor nástrojov na ladenie výkonu aplikácií Sun Studio Performance Tools [10]. Sú súčasťou vývojového prostredia Sun Studio, ktoré je dostupné zadarmo po registrácii na stránkach Sun Developer Network pre systémy Sun Solaris a Linux. Na svoj beh používa prostredie Java, na pozadí spúšťa konzolové utility na

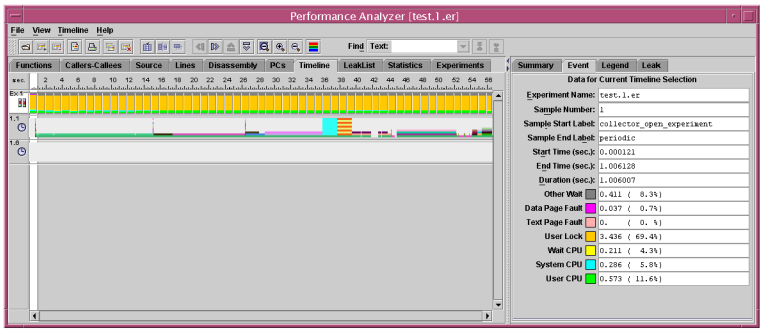
zbieranie a vyhodnocovanie analyzovaných dát. Podporuje štandardné vlákna POSIX, Solaris threads aj štandard OpenMP (vo svojom prekladači). Profilovanie aplikácií na základe hardvérových počítadiel je podporené hlavne pre procesory SPARC. Okrem toho aj procesorov x86 a AMD Opteron (na procesore Intel Centrino Duo sa mi meranie nepodarilo vykonať, na procesore AMD Athlon XP sa spustilo).



Obrázek 4.3: Sun Studio Performance Tools výkonová analýza kódu.

Jedinečnou vlastnosťou, deklarovanou spoločnosťou Sun, je značenie dátových štruktúr pri analýze výpadkov pamäte. Vývojár by tak mal byť schopný určiť pôvod výpadku v dátových štruktúrach, nie iba v inštrukcii.

Výkonová analýza vlákien (obr. **??**) je podľa môjho názoru jednoduchšie spracovaná, než u nástroja Intel Thread Profiler. Neumožňuje takú podrobnú analýzu priebehu vykonávania vlákien ako s rôznymi pohľadmi u aplikácie Thread Profiler.

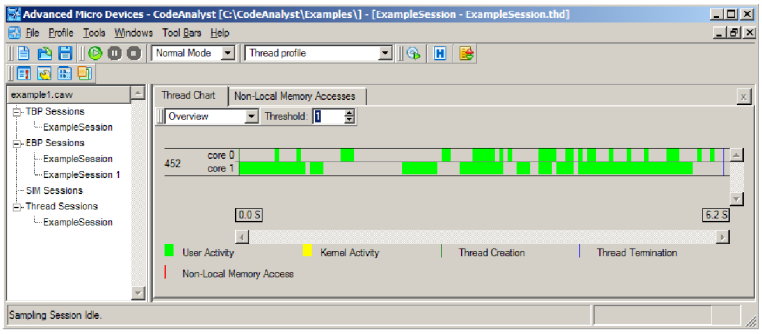


Obrázek 4.4: Sun Studio Performance Tools výkonová analýza vlákien.

Na statickú detekciu uviaznutí a prístupových konfliktov je určená samostatná utilita **lock_lint** [8], ktorá nemá grafické užívateľské rozhranie. Práca s ňou je tým pádom pravdepodobne komplikovanejšia, než s programom Intel Thread Checker. Sun Studio Performance Tools ani lock_lint som podrobnejšie neskúšal, väčšinu času mi zabrala inštalácia prostredia a riešenie s tým súvisiacich problémov (nepodporovaná distribúcia openSUSE, nedostatok pamäte pre inštaláciu Solaris Express, Developer Edition, pravdepodobne nepodporovaný typ procesoru od firmy Intel). Kolekcia nástrojov pre výkonovú analýzu od Sun Microsystems je zrejme zameraná viac na procesory SPARC tejto firmy a serverové aplikácie, nie desktopové.

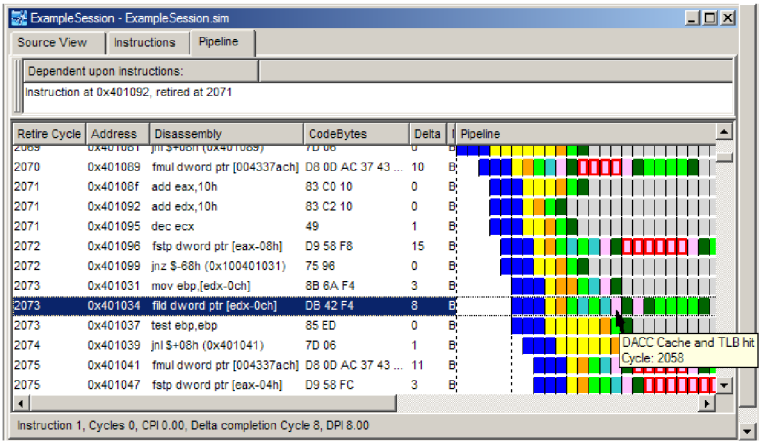
AMD CodeAnalyst Performance Analyzer

Reakciou firmy AMD na ladiace nástroje Intelu je aplikácia AMD CodeAnalyst Performance Analyzer [4]. Je to logický krok, keďže procesory AMD nie sú softvérom od Intelu podporované a sú tak v menšej nevýhode. Riešenie od AMD je však jednoduchšie a bez mnohých vlastností nástrojovej sady Intel Threading Tools. Umožňuje zbieranie výkonových dát pomocou systémového alebo hardvérového časovača a udalostí (podobne ako všetky ostatné nástroje), tj. frekvencia vykonávania riadkov kódu, pamäťové výpadky, nesprávne predpovedané skoky a pod. Možnosti nastavení sú však značne obmedzené. Obrázok 4.3 ukazuje výsledok výkonovej analýzy vlákien programom CodeAnalyst.



Obrázek 4.5: AMD CodeAnalyst profilovanie vlákien.

Jedinečnou vlastnosťou nástroja však je možnosť simulácie procesorového reťazca pre procesory AMD Athlon, Athlon XP, Opteron a Athlon 64. Programátor si v okne s výpisom zdrojového kódu vyznačí úsek, pre ktorý chce zobraziť jednotlivé fázy reťazca a po spustení sa nazhromaždia požadované informácie pre daný úsek.



Obrázek 4.6: AMD CodeAnalyst simulácia procesorového reťazca.

Aplikácia AMD CodeAnalyst je dostupná aj vo verzii pre operačný systém Linux, ktorá používa modifikovanú verziu už spomínaného programu OProfile. Analýzu výkonu týmto nástrojom (Windows aj Linux verzia) nie je možné vykonať bez administrátorských práv (podobne je to s mnohými meraniami programom VTune Performance Analyzer).

Kapitola 5

Násobenie matíc

Rozdiel medzi rýchlosťou procesoru a pamäte neustále narastá. Pre vyváženie tohto rozdielu sú v počítačových systémoch používané pamäťové hierarchie [2]. Pomalá hlavná pamäť má veľkú kapacitu a je lacná. Nad ňou sa nachádza niekoľko vrstiev vyrovnávacích pamätí, ktoré sú rýchlejšie, ale podstatne menšie pre ich vysokú cenu. Tieto vyrovnávacie pamäte sa nachádzajú vo vnútri procesoru a sú typicky dvojúrovňové, tzn. jedna rýchlejšia, menšia a jedna pomalšia, väčšia. V prípade viacprocesorového systému má každý procesor svoju vlastnú vyrovnávaciu pamäť. Viacjadrové procesory môžu mať vyrovnávaciu pamäť zdieľanú. Pri požiadavke procesora na načítanie pamäťovej adresy, ktorá sa nenachádza v bezprostrednej vyrovnávacej pamäti procesoru sa hľadá vo vyrovnávacej pamäti druhej úrovne. Ak sa nepodarilo dáta nájsť ani v nej a vyrovnávaciu pamäť tretej úrovne systém neobsahuje, načítanie prebehne po systémovej zbernici z hlavnej pamäte. Operácia načítania dát z hlavnej pamäte však už trvá rádovo stovky taktov procesoru, čo už znamená veľké zdržanie výpočtu. Ak je vyrovnávacia pamäť preplnená a procesor požaduje načítanie nového bloku pamäte, ktorý sa v nej nenachádza, prichádza na rad odstránenie jedného z načítaných blokov pamäte. Existuje niekoľko stratégií, ako vybrať blok vyrovnávacej pamäte na uvoľnenie. Najčastejšie je to odstránenie najstaršieho a najmenej používaného bloku.

Po načítaní dátového bloku do vyrovnávacej pamäte je teda výhodné, ak v nej zotrvá čo najdlšie, pričom procesor ho využije na výpočet v čo najväčšej miere, aby ho po odstránení z vyrovnávacej pamäte nemusel opäť načítať. Tomuto princípu sa hovorí *lokalita odkazov*. Aby sa pri násobení matíc dosiahla čo najväčšia miera lokality odkazov, je potrebné maticu mierne reorganizovať. Matica je dvojrozmerný útvar, pričom pamäť je iba lineárne adresovateľná. Naivné usporiadanie prvkov matíc je naukladanie riadkov vedľa seba. Tento spôsob uloženia prvkov je podporený aj v programovacom jazyku C. Počas násobenia sa načíta jeden riadok a jeden stĺpec matice, ich hodnoty sa prvok po prvku vynásobia a výsledky sa sčítajú do jedinej hodnoty, ktorá sa ukladá do výslednej matice. S lokalitou odkazov pri načítaní riadku nie je problém, keďže načítaný blok pamäte bude s veľkou pravdepodobnosťou obsahovať prvky toho istého riadku. Problém nastáva pri načítaní prvkov stĺpca, najmä vo veľkých maticiach. Ak je riadok matice veľmi dlhý a nezmestí sa do jedného bloku načítaného do vyrovnávacej pamäte, načítanie prvkov stĺpca bude znamenať načítanie celého bloku vyrovnávacej pamäte pre využitie jediného čísla (napríklad štvorbajtového desatinného čísla). Už na prvý pohľad to nevyzerá ako optimálne riešenie.

Dôvtipmejšou organizáciou prvkov matice je jej „rozbitie“ na menšie matice, ktoré sú optimalizované pre veľkosť vyrovnávacej pamäte a tak sú naukladané vedľa seba. Celkové násobenie matíc sa potom uskutočňuje ako násobenie a sčítanie riadkov a stĺpcov veľkej matice obsahujúce bloky menších matíc, akoby boli jej prvkami.

Najmenšou dátovou jednotkou, ktorá sa načíta do vyrovnávacej pamäte je tzv. *cache line*. Je to blok dát, ktorý má typicky niekoľko desiatok bajtov. Pre procesor, na ktorom bola úloha ladená, bola veľkosť bloku vyrovnávacej pamäte rovná 64 bajtov. K dosiahnutiu zarovnania menších matíc na bloky vyrovnávacej pamäte sa ukladajú ich prvky prekladane. To znamená, že podľa zvoleného koeficientu prekladania sa za seba ukladajú prvky v matici pod sebou a potom ďalšia skupina prvkov v matici napravo od predošlých. Tento postup by nebol účinný, keby bol ukazovateľ alokovanej pamäte náhodný, preto je zarovnávaný na veľkosť bloku vyrovnávacej pamäte, pričom aj veľkosť alokovanej pamäte je zarovnávaný.

Samotná paralelizácia úlohy spočíva vo vytvorení dvoch ďalších vlákien, ktorým je odovzdaná štruktúra so všetkými potrebnými informáciami k spusteniu výpočtu nezávislého bloku dát. Vlákna si podľa identifikačného čísla sami vypočítajú riadky blokov matíc, ktoré majú na starosti. V závere, po ukončení výpočtu, hlavné vlákno programu počká dobiehajúce výpočtové vlákna. Implementácia správy vlákien využíva knižnicu POSIX threads [9], ktorá bola použitá namiesto plánovaného OpenMP [1]. POSIX threads boli zvolené pre nedostupnosť prekladača podporujúceho OpenMP, ktorý by vyhovoval ladiacemu nástroju Intel Thread Profiler. Ten vyžaduje vloženie špeciálnych symbolických inštrukcií do binárneho kódu programu. Prekladač Intel C++ Compiler nebol z licenčných dôvodov dostupný a Microsoft C++ Compiler bol dostupný len vo verzii bez podpory OpenMP.

5.1 Popis implementácie

Program pre násobenie matíc môže byť spustený v niekoľkých režimoch práce, podľa zadaných argumentov príkazu pri spustení. Aplikácia teda môže byť spustená ako:

1. Jednovláknový výpočet bez optimalizácie pamäťového prístupu pre pamäťovú hierarchiu. Tento režim je implementovaný pre testovacie účely, aby bolo možné porovnať výsledné matice (najmä väčších rozmerov).

2. Viacvláknový výpočet bez optimalizácie pamäťového prístupu. Slúži na porovnanie výkonu s paralelnou variantou algoritmu so zapnutou optimalizáciou lokality odkazov.

3. Viacvláknový výpočet s organizáciou vstupných matíc do menších blokov dvoch úrovní podľa dostupných rýchlych pamätí procesoru. Táto varianta bola cieľom práce a mala by dosahovať lepší výkon než predchádzajúce (výsledky budú zhodnotené na záver).

Každý jeden režim je identifikovaný podľa zadaných argumentov. Tie sú načítané po spustení programu. Zoznam použiteľných parametrov, ktorými sa nastavujú vlastnosti výpočtu, obsahuje tabuľka 5.1.

Vo vstupnom súbore musia prvé tri čísla, oddelené bielymi znakmi, udávať rozmery vstupných matíc. Prvé číslo je brané ako počet riadkov prvej matice, druhé číslo definuje počet stĺpcov prvej matice a súčasne počet riadkov druhej matice a tretie číslo určuje počet stĺpcov druhej matice. Za týmito tromi číslami musí nasledovať zoznam prvkov matíc po riadkoch. Rozmery vypočítanej matice sú odvodené od prvého a tretieho čísla, udávajúcich rozmery vstupných matíc.

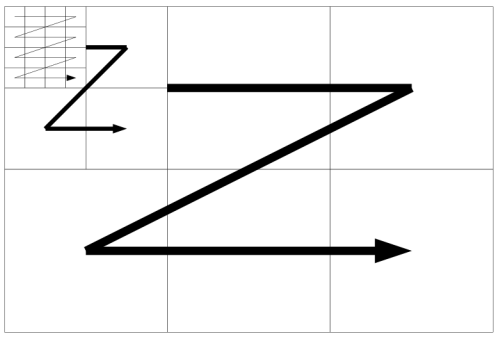
V prípade, že je použitý výpočet bez optimalizácie usporiadania matíc (t.j. 1 alebo 2), načítajú sa matice do poľa po riadkoch. Veľkosť poľa je odvodená od rozmerov matice, ktoré sú zarovnané na veľkosť blokov pre lepšiu prácu s vyrovnávacou pamäťou, aj keď blokové usporiadanie nie je využité. Ľahšie sa potom porovnávajú výsledky, v ktorých sú matice doplnené vpravo a nadol nulami.

Názov	Formát	Povinný
Vstupný súbor	`<názov_súboru>`	áno
Výstupný súbor	`-o <názov_súboru>`	nie
Počet vláken	`-t <počet_vláken>`	nie
Usporiadať matice do blokov	`-C`	nie
Veľkosť "cache line" v bajtoch	`-cl <veľkosť_cache_line>`	nie
Rozmery bloku úrovne L1	`-l1 <počet_riadkov> <počet_stĺpcov>`	nie
Rozmery bloku úrovne L2	`-l2 <počet_riadkov> <počet_stĺpcov>`	nie

Tabulka 5.1: Zoznam parametrov aplikácie.

Ak nie je zadaný parameter určujúci počet vlákien, automaticky sa použije varianta algoritmu bez paralelizmu. V tomto prípade sa riadky jednej matice vynásobia so stĺpcami druhej matice a výsledok sa uloží. Podobne je to aj v prípade, že počet vláken je zadaný s tým rozdielom, že riadky prvej matice sú rovnomerne pridelené jednotlivým procesorom, ktoré potom násobia im priradené riadky so všetkými stĺpcami druhej matice. Každý procesor ukladá výsledky do jemu prislúchajúceho riadku výstupnej matice, takže k dátovému konfliktu nemôže dôjsť a nie je potrebná ani žiadna synchronizácia.

Posledná tretia varianta algoritmu načíta matice rovnakým spôsobom ako prvé dve, ibaže po načítaní matice preusporiada do blokov podľa zadaných parametrov programu. Pre blokové rozloženie prvkov matice najpr alokuje pamäť, ktorej ukazovateľ zarovná na veľkosť „cache line“. Zabezpečí sa tým efektívnejšie uloženie prvkov matice, pretože každý bajt tohto najmenšieho načítaného bloku pamäte bude pri výpočte využitý. Je vhodné, aby bola veľkosť bloku úrovne L1 deliteľná týmto číslom, inak nebude usporiadanie optimálne. Po alokovaní pamäte prekopíruje prvky vstupnej matice do nového poľa, v ktorom sú prvky ukladané striedavo tak, že za sebou sa nachádzajú úseky niekoľkých riadkov matice. Tento systém je dvojúrovňový, aby sa lepšie prispôsobil hierarchii vyrovnávacích pamätí procesoru. Zobrazený je na nasledujúcom obrázku:



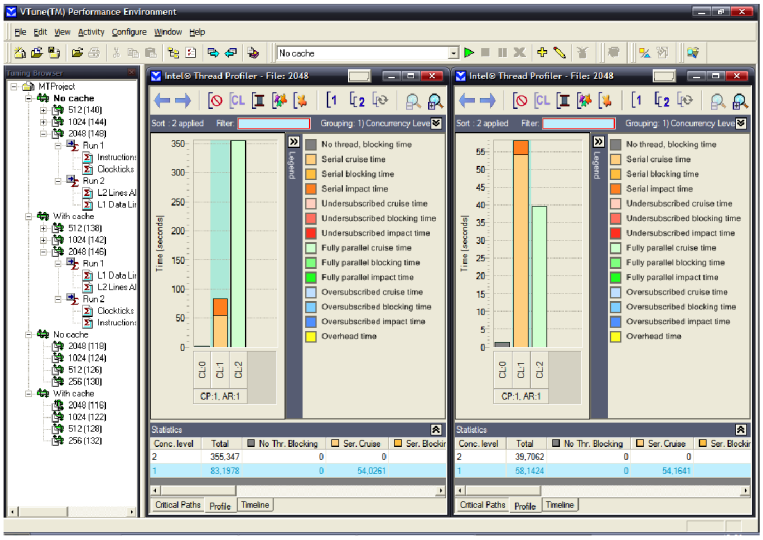
Obrázek 5.1: Systém blokového usporiadania matíc.

Šípky ukazujú postupnosť prvkov ako sú za sebou ukladané v alokovanom poli. Násobenie takýchto matíc je potom rozdelené na násobenie menších blokov matíc medzi sebou, akoby to boli prvky matice. Bloky matíc na najnižšej úrovni sú už násobené skutočne po prvkoch. Rozdelenie medzi vlákna je potom realizované pridelením riadkov blokových matíc úrovne

L2 procesorom. Opäť ako pri neblokovej variante nedochádza ku konfliktom zápisu, takže každý procesor môže venovať plný výkon jemu pridelenej časti úlohy.

5.2 Namerané výsledky

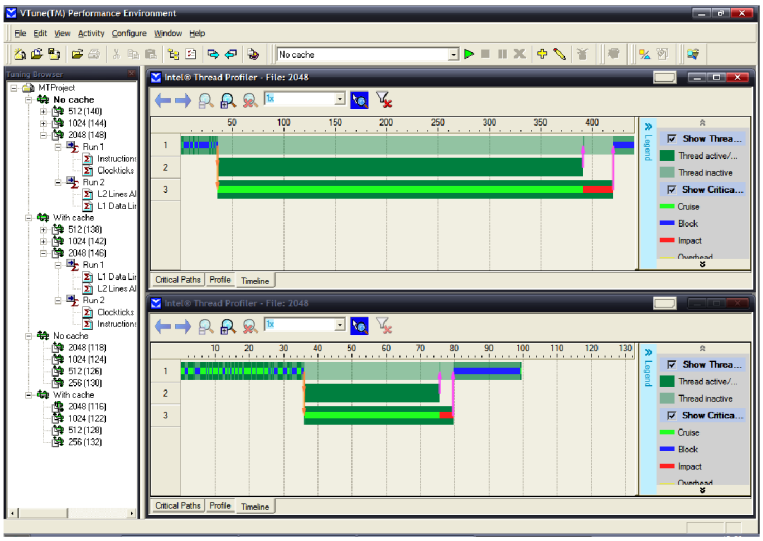
Za účelom merania výkonu naprogramovanej aplikácie boli vytvorené vstupné súbory obsahujúce matice rôznych rozmerov. Matice sú štvorcové (program dokáže násobiť aj matice nerovnakých rozmerov) s počtom riadkov 256, 512, 1024, 2048. Prvky matíc sú celé čísla v rozpätí od 0 do 100, generované náhodne. Takto pripravené testovacie dáta boli nastavené ako vstupy pre jednotlivé výpočty analyzované kolektorom Thread Profiler aplikácie VTune Performance Analyzer. Pre každú variantu algoritmu (s optimalizáciou a bez nej) som vytvoril tzv. **Activity**, ktorú som spustil vždy s iným vstupným súborom. Z nameraných dát som si všímal najmä dĺžku kritickej cesty v **Critical Paths** pohľade, ktorá udáva celkové trvanie výpočtu. Ďalej som prepol na pohľad **Profile**, z ktorého som vyčítal trvanie paralelnej časti algoritmu. Tento pohľad je zobrazený aj na obrázku 5.2.



Obrázek 5.2: Rozloženie časových kategórií trvania výpočtu.

Pohľad **Timeline** (obr. 5.2) graficky zobrazuje časový priebeh výpočtu, kde vláknam prislúchajú jednotlivé riadky a striedavo sa po nich vynie kritická cesta. Zelenou farbou je vyznačená v prípade, keď žiadne vlákno nemusí čakať na to druhé, kým dokončí výpočet. Červená farba sa v našom grafe nachádza preto, že po skončení úlohy vlákna čakajú navzájom na seba (príkaz `pthread_join`), až potom sa ukončí program. Modrá farba značí blokovanie vlákna, napr. vstup-výstupnou operáciou akou je načítavanie matíc z pevného disku počítača. Uvedený obrázok je výsledkom násobenia matíc rozmerov 2048 a je vidieť, že optimalizovaná varianta algoritmu vykonala úlohu niekoľko násobne rýchlejšie, než jednoduchá implementácia.

Konkrétne namerané hodnoty obsahujú tabuľky 5.2 a 5.3. Sú v nich celkové časy jednotlivých variant ako aj časy súbežného vykonávania vlákien. Všetky časy sú uvedené v sekundách. Ako vidieť, algoritmus s blokovým usporiadaním matíc vykoná výpočet rýchlejšie, než algoritmus s lineárnym rozložením prvkov matice. Optimalizovaná varianta vidi-



Obrázek 5.3: Graf časového priebehu súbežných vlákien.

teľne dosahuje vyšší pomer sekvenčného času oproti paralelnému, čo je spôsobené predspracovaním matíc. To však s prehľadom vykompenzuje pri samotnom paralelnom násobení.

Rozmer matice	256	512	1024	2048
Celkový čas v sek.	1.13	4.46	59.36	440.63
Trvanie paralelného výpočtu	0.063	0.91	45.05	355.35

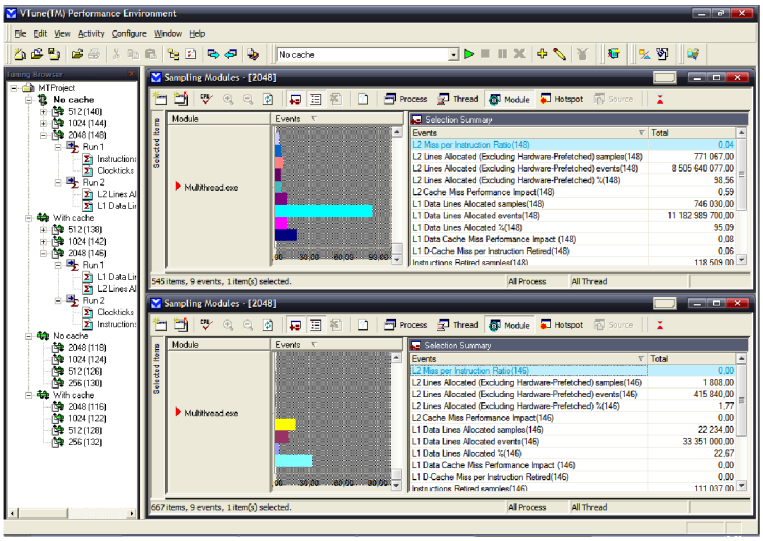
Tabulka 5.2: Namerané časy pre zjednodušený algoritmus.

Rozmer matice	256	512	1024	2048
Celkový čas v sek.	1.13	4.13	18.93	99.32
Trvanie paralelného výpočtu	0.008	0.61	4.88	39.71

Tabulka 5.3: Namerané časy pre optimalizovaný algoritmus.

Po analýze programu kolektorom Thread Profiler som pridal ďalšie **Activity**, ktorým som nastavil systémové kolektory štandardné pre aplikáciu VTune Preformance Analyzer. Pridal som tzv. **Sampling** kolektor, ktorý zbiera informácie o vybraných udalostiach v systéme, napr. počet alokácií blokov vyrovnávacej pamäte L2, prípadne L1, počet vykonaných inštrukcií, počet pozastavených inštrukcií (napr. pre výpadok pamäte) a pod. Je možné zvoliť aj množstvo iných sledovaných parametrov procesoru. Medzi inými aj z kategórie zbernicových udalostí, DTLB udalostí, štatistík skokových inštrukcií, štatistík inštrukcií s číslami s pohyblivou desatinnou čiarkou a mnoho iných. Zvolené sú uvedené v tabuľkách 5.4 a 5.5 v prvom stĺpci. Namerané hodnoty týchto udalostí a štatistických priemerov sa nachádzajú v ďalších stĺpcoch. Meranie prebiehalo pre každý vstup v niekoľkých fázach. Najprv sa spúšťal výpočet pre získanie jednej skupiny hodnôt (štatistiky inštrukcí), potom kalibrácia pre druhú skupinu hodnôt (štatistiky vyrovnávacích pamätí) a samotné získavanie dát. Tento postup sa vykonával samostatne zvoleným kolektorom. Výstup bol

zobrazený vo forme ako na obrázku 5.2. Naľavo je graf nameraných hodnôt (každý riadok je zvlášť škálovaný, takže nemá priveľkú výpovednú hodnotu), napravo sú konkrétne čísla meraných udalostí a štatistických informácií. Dole je legenda grafu so škálovacím faktorom a inými parametrami zobrazenia.



Obrázek 5.4: Zobrazenie nameraných systémových dát.

Udalosť	512	1024	2048
Alokácií L2 bloku	346 626	1 043 987 494	8 505 640 077
Dopad výpadkov L2 na výkon	0.01	0.60	0.59
Alokácií L1 bloku	134 812 674	1 074 818 088	11 182 989 700
Dopad výpadkov L1 na výkon	0.34	0.06	0.08

Tabulka 5.4: Štatistika vyrovnávacích pamätí pre zjednodušený algoritmus.

Udalosť	512	1024	2048
Alokácií L2 bloku	9 042	60 291	415 840
Dopad výpadkov L2 na výkon	0.00	0.00	0.00
Alokácií L1 bloku	609 234	4 373 187	33 351 000
Dopad výpadkov L1 na výkon	0.00	0.00	0.00

Tabulka 5.5: Štatistika vyrovnávacích pamätí pre optimalizovaný algoritmus.

Hodnoty boli namerané na počítači s procesorom Intel Centrino Duo s dvomi jadrami, ktorého „cache line“ má veľkosť 64B, veľkosť vyrovnávacej pamäte úrovne L2 je 2MB a veľkosti dvoch dátových vyrovnávacích pamätí úrovne L1 sú 32kB. Parametre programu som teda volil nasledovne. Pre maticové bloky úrovne L1 som zvolil veľkosť 64 riadkov a 32 stĺpcov, čo pri veľkosti prvku 4B (32-bitové celé čísla) znamená 8kB použitej pamäte pre jednu maticu. Keďže potrebujeme mať súčasne v pamäti tri matice (prvú a druhú vstupnú

a jednu výslednú), budú spolu zaberať 24kB, čo sa do vyrovnávacej pamäte úrovne L1 zmestí aj s rezervou. Matice týchto rozmerov sú ďalej usporiadané do blokov úrovne L2, pre ktoré som zvolil výšku 4 a šírku 8 blokov úrovne L1. To spolu dáva veľkosť maticového bloku úrovne L2 256kB, čo pre tri matice umiestnené v pamäti znamená 768kB. Keďže vyrovnávacia pamäť úrovne L2 je zdieľaná, zvolil som menšie bloky, aby každému procesoru stačila veľkosť 1MB. Spustenie programu aj s vypočítanými parametrami teda vyzeralo nasledovne:

```
./Multithread.exe input.txt -o output.txt -t 2 -C -cl 64 -l1 64 32 -l2 4 8
```

Tento zápis obsahuje aj priložený `Makefile`, ktorý sa spustením `make && make run` vykoná (pre cieľ `make linux` je potrebné tento zápis pozmeniť – odstrániť príponu).

Alokácie L2, prípadne L1, bloku znamenajú, že došlo k výpadku pamäte a procesor potreboval alokovať nový blok vo vyrovnávacej pamäti pre načítanie dát. Veľkosť týchto blokov je závislá na implementácií procesoru a nerovná sa veľkosti blokov matíc. Druhá meraná veličina, dopad výpadkov L2, prípadne L1, na výkon je vypočítaná aplikáciou VTune Performance Analyzer z počtu výpadkov vyrovnávacej pamäte a počtu spotrebovaných taktov procesoru vzťahom [12]:

$$\frac{80 \times (alokovanych\_L2\_blokov)}{(taktov\_procesoru)}$$

pre výpadok vo vyrovnávacej pamäti L2 a pre výpadok vo vyrovnávacej pamäti L1 platí:

$$\frac{8 \times (alokovanych\_L1\_blokov)}{(taktov\_procesoru)}$$

Podľa [12] je dobré dosiahnuť hodnoty menej než 0.01 pre L2 a menej než 0.05 pre L1. Za neprijateľné sú považované hodnoty viac ako 0.28 pre L2 a viac ako 0.3 pre L1. Z tabuľky pre zjednodušenú variantu algoritmu 5.4 je vidieť problém tohto prístupu k implementácii bez zreteľa na lokalitu odkazov. Pre matice zaberajúce 1MB (s počtom riadkov 512) je najväčším problémom počet výpadkov na úrovni L1, keďže matice sa ako-tak vmestia do vyrovnávacej pamäte L2. Pri väčších maticiach je z počtu výpadkov L2 a L1 zrejmé, že každý výpadok L1 viedol tiež na výpadok v L2. To mohlo byť následkom toho, že pri počte stĺpcov 1024 sú prvky v matici pod sebou rozmiestnené s rozostupom 4kB. Každý prvok stĺpca teda alokuje nový blok vyrovnávacej pamäte L2. Pri množstve riadkov 1024 to však znamená načítanie 4MB dát do L2, niektoré bloky teda musia byť vyhodené. Sú to pravdepodobne tie najstaršie čítané, takže procesor vždy cyklicky vyhadzuje načítané bloky z vyrovnávacej pamäte. Pri počte riadkov 2048 je tento efekt iba umocnený. Naproti tomu sú výsledky pre optimalizovaný algoritmus z tabuľky 5.5 priam ideálne. Systém rozdelenia matice do menších blokov teda prispel k výraznému zvýšeniu efektivity algoritmu, najmä v prípade násobenia matíc veľkých rozmerov.

Kapitola 6

Záver

Hlbšie pochopenie princípov fungovania vlákien v systéme je kľúčom k pochopeniu problémov, ktoré môžu vzniknúť počas vývoja viacvláknovej aplikácie. Preto boli úvodné kapitoly venované podrobnejšiemu výkladu mechanizmov zabezpečujúcich súčasný beh viacerých vlákien vrátane poodhalenia ich koreňov v nedávnej minulosti. Aj spôsob, akým sú hardvérové prostriedky implementované, do značnej miery ovplyvňuje k akým problémom môže z pohľadu softvéru dôjsť. Príkladom takejto závislosti sú optimalizácie vykonávania inštrukcií, keď sú inštrukcie programu vykonávané mimo poradia. Môže dôjsť k situácii, keď načítanie hodnoty z pamäte v jednom vlákne bude vykonané skôr, napríklad pred zápisom do pamäte na inej adrese, než by to programátor v druhom vlákne očakával.

Rovnako dôležité je pochopiť princípy prostriedkov ponúkaných programovacími prostrediami na zvládnutie synchronizácie, správy toku riadenia a komunikácie medzi vláknami, aby bolo možné na základe konkrétnych požiadaviek riešenej úlohy použiť najvhodnejší mechanizmus. Nesprávne použitie týchto prostriedkov môže spôsobiť zbytočné zníženie výkonu aplikácie.

Ako demonštračnú úlohu som implementoval algoritmus násobenia matíc, u ktorého som optimalizoval lokalitu odkazov usporiadaním prvkov matíc do blokov dvoch úrovní. Prvá úroveň zefektívňuje využitie vyrovnávacej pamäte prvej úrovne, dostupnej na procesoroch, druhá úroveň vylepšuje využitie vyrovnávacej pamäte druhej úrovne. Po naprogramovaní algoritmu som vytvoril testovaciu implementáciu jednoduchého násobenia matíc (teda bez optimalizácie). Výkon tejto verzie som porovnával s výkonom vylepšenej varianty. Merania časov som vykonával aplikáciou Intel Thread Profiler, na merania výpadkov vo vyrovnávacích pamätiach som použil nástroj Intel VTune Performance Analyzer. Výsledkom bolo niekoľko násobné urýchlenie násobenia matíc vďaka podstatnému zlepšeniu práce s pamäťovou hierarchiou.

V ďalších krokoch som vytvoril krátky návod na prácu s aplikáciami Thread Profiler a Thread Checker, ktorý pomocou príkladov nachádzajúcich sa v inštalačnom adresári nástrojov sprevádza používateľa krok za krokom pri vytvorení projektu, vykonaní zberu dát a analýze výsledkov.

Na záver som sa poobhliadol po ďalších nástrojoch rovnakého zamerania, ktorých vlastnosti som porovnával s možnosťami softvérového balíku od firmy Intel. Po vyskúšaní aplikácií Thread Profiler a Thread Checker môžem skonštatovať, že je to najlepší softvér svojho druhu na trhu. Jeho použitie je po istom čase plne intuitívne, je založený na koncepte kolektorov (modulov, ktoré zbierajú a vyhodnocujú istú skupinu dát – aj Thread Profiler a Thread Checker sú iba kolektory aplikácie VTune Performance Analyzer). Ponúka nespočetné možnosti analýzy činnosti programu, aj systému ako celku (diskové operácie,

sieťový prenos, transakcie systémovej zbernice atď.). Jeho využitie na výukové účely môžem iba odporúčať, i keď realizácia môže naraziť na technické problémy. Niektoré kolektory totiž vyžadujú administrátorské práva pre prístup do operačného systému. Thread Profiler a Thread Checker nie sú tie prípady, merania uskutočňujú pomocou modifikácií binárneho súboru programu. Používateľ však prichádza napríklad o prehľad výpadkov vo vyrovnávacích pamätiach, ktoré bez vyšších práv nie je možné analyzovať (aplikácia neodpovedá). Nástroje VTune Performance Analyzer a Thread Checker sú ale vo verzii pre operačný systém Linux dostupné zadarmo pre osobné a nekomerčné účely. Ak má teda niekto prístup k počítaču s procesorom Intel (nutná podmienka) s nainštalovaným operačným systémom Linux, nič mu nebráni vyskúšať si tento softvér.

Zvyšovanie počtu jadier v mikroprocesoroch je podľa všetkého smer, ktorým sa ich výrobcovia vydali, aby tak udržali neustály rast výkonu. Záleží už len na schopnostiach softvérových vývojárov, či dokážu ponúkaný výkon naplno využiť. Keďže je práca zameraná priamo na analýzu výkonu a problémov spojených s využitím viacvláknového paralelizmu, pokladám ju za veľký osobný prínos najmä z pohľadu budúceho využitia poznatkov, keď počty jadier v procesoroch bežne presiahnu štvoricu a ich potenciálny výkon nebude môcť byť softvérovými vývojármi prehliadaný.

Literatura

[1] V. Dvořák. *Architektura a programování paralelních systémů.* VUTIUM Brno, 2004. ISBN 80-214-2608-X.

[2] Nadav Eiron. Matrix multiplication: A case study of enhanced data cache utilization. `http://www.theeirons.org/Nadav/pubs/MatrixMult.pdf`.

[3] Jason Robers Shameem Akhter. *Multi-Core Programming.* Intel Press, 2006. ISBN 0-9764832-4-6.

[4] WWW stránky. Amd codeanalyst performance analyzer. `http://developer.amd.com/cawin.jsp`.

[5] WWW stránky. Intel thread checker online help. `http://www.intel.com/cd/software/products/asmo-na/eng/300225.htm`.

[6] WWW stránky. Intel thread profiler online help. `http://www.intel.com/cd/software/products/asmo-na/eng/300638.htm`.

[7] WWW stránky. Introduction to parallel computing. `http://www.llnl.gov/computing/tutorials/parallel_comp/`.

[8] WWW stránky. Lock_lint - static data race and deadlock detection tool for c. `http://developers.sun.com/sunstudio/articles/locklint.html`.

[9] WWW stránky. Posix threads programming. `http://www.llnl.gov/computing/tutorials/pthreads/`.

[10] WWW stránky. The sun studio performance tools. `http://developers.sun.com/sunstudio/articles/perftools.html`.

[11] WWW stránky. Visual threads technical information. `http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,5231,00.html`.

[12] WWW stránky. Vtune performance environment help. `http://www.intel.com/cd/software/products/asmo-na/eng/342505.htm`.

Stručný návod k Intel Threading Tools

Intel Thread Profiler

Preklad ukážkového kódu

Inštalácia aplikácie Intel Thread Profiler obsahuje niekoľko ukážkových programov, ktoré sa vo forme zdrojových kódov nachádzajú v inštalačnom adresári. Ako prvý príklad sa tu nachádza implementácia generátora prvočísel (`Primes`), ktorý používa štandardné API pre vlákna operačného systému Windows. Program hľadá a zaznamenáva prvočísla od jednotky až po 100000 testovaním deliteľnosti nových čísel bezozvyšku predchádzajúcimi prvočíslami. Program pre tento účel vytvorí štyri vlákna. V ďalších krokoch použijete aplikáciu Thread Profiler na vyhodnotenie výkonu a nájdenie miest v zdrojových kódoch programu, kde je možné zlepšiť využitie procesoru.

Preklad kódu:

1. Otvorte súbor pracovnej plochy vývojového prostredia Microsoft Visual Studio pre pripravený projekt `Primes.dsw`, obvykle umiestnený v adresári aplikácie Thread Profiler:
   `C:\Program Files\Intel\VTune\tprofile\samples\Primes`

2. Preložte načítaný projekt. Tento projekt obsahuje nasledujúce nastavenia: `/Zi` pre vloženie symbolov, `/Od` pre vypnutie optimalizácií, `/fixed:no` nastavenie zostavovacieho programu pre vytvorenie premiestniteľného kódu a `/MDd` alebo `/MTd` pre preklad s použitím pre vlákna bezpečných knižníc. Všetky tieto nastavenia sú dôležité pre aplikáciu Thread Profiler, aby mohla poskytnúť podrobný prehľad, napr. o názvoch premenných a číslach riadkov, na ktorých je potrebné vylepšiť výkon. Výsledkom prekladu je program `Primes.exe` so zahrnutými potrebnými informáciami pre ladenie.

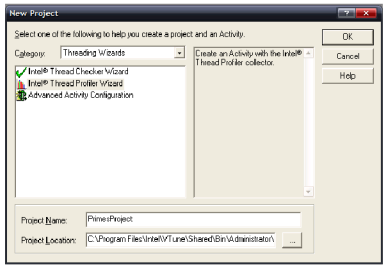
Zbieranie dát

Sprievodca Intel Thread Profiler Wizard umožňuje jednoduchým spôsobom vygenerovať a nazhromaždiť informácie o priebehu a výkone viacvláknovej aplikácie. Pre použitie sprievodcu sú potrebné nasledujúce kroky:

1. Spustite Intel Thread Profiler nájdením príslušného odkazu v ponuke Štart operačného systému Windows alebo dvojitým kliknutím na ikonu aplikácie Intel VTune Performance Analyzer nachádzajúcej sa na ploche.

2. Následne zvoľte v dialógovom okne **Easy Start** alebo na nástrojovej lište aplikácie voľbu **New Project**, po ktorej sa Vám objaví dialóg **New Project**.

3. V roletovom výbere **Category** zvoľte **Threading Wizards** a vyberte **Intel Thread Profiler Wizard**.



4. Pomenujte projekt, napr. `PrimesProject`. Aplikácia automaticky doplní adresár na uloženie projektu **Project Location**, ktorý je možné v prípade potreby zmeniť.

5. Stlačte **OK**. Otvorí sa sprievodca **Intel Thread Profiler Wizard**.

6. Zo zoskupenia **Threading type** vyberte **Threaded (Windows* API or POSIX* threads)**, keďže ukážkový príklad používa Windows* API.

7. Pod označením **Launch an application** kliknite na **[...]** a v otvorenom dialógovom okne vyhľadajte spustiteľný súbor `Primes.exe` preložený pre ladiace účely. Thread Profiler štandardne doplní položku **Working directory** adresárom, v ktorom sa program nachádza.

8. Kliknite na **Finish** pre dokončenie sprievodcu. Thread Profiler najprv poupraví aplikáciu, vykoná ju, nazbiera dáta a následne ich zobrazí.

Presné výsledky sú závislé na konfigurácii Vášho systému, ale pravdepodobne uvidíte zvislé obdĺžniky s farebným rozdelením. V tomto bode je už možné ďalej pokračovať analýzou kritických miest aplikácie, ktoré znižujú výkon viacvláknového vykonávania.

Analýza výsledkov a oprava kódu

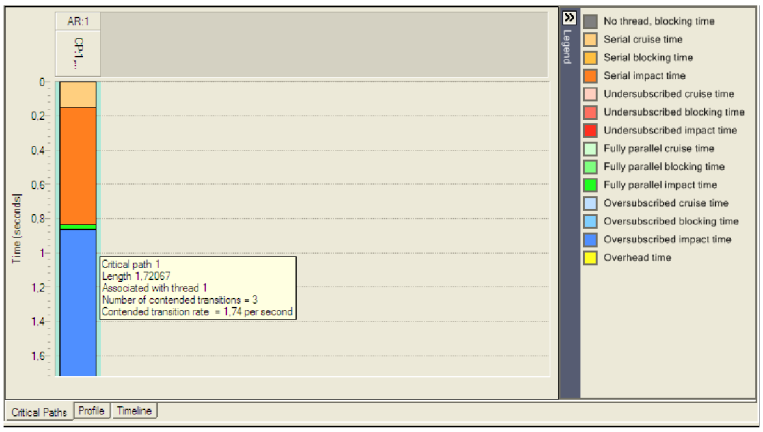
V tomto odseku sa oboznámite s niektorými pohľadmi aplikácie Thread Profiler, ktoré umožňujú identifikovať výkonové problémy týkajúce sa behu viacerých vlákien. Medzi tieto pohľady patria **Critical Paths**, **Profile** a **Timeline**. Následne budete schopní zvážiť potrebné zásahy do kódu aplikácie, ktoré povedú k zvýšeniu výkonu.

Critical Paths pohľad

Thread Profiler sleduje priebeh všetkých vlákien v aplikácii. Najdlhšia spojitá cesta vedúca skrz pribehy sa nazýva kritická cesta. Pohľad **Critical Paths** nám ukazuje spôsoby využitia času výpočtu na kritickej ceste programu.

Na nasledujúcom obrázku sú výsledky nazbierané po spustení programu na dvojjadrovom procesore Intel Centrino Duo:

Presuňte kurzor myši nad obdĺžnikové znázornenie rozloženia kritickej cesty, čím sa Vám zobrazí text so zhrnutím zobrazených dát. Z obrázku vyššie je možné vyčítať, že kritická cesta dosiahla celkovú dĺžku niečo pod dve sekundy.



Časové kategórie

Thread Profiler rozdeľuje čas do niekoľkých kategórií, reprezentovaných v grafe odlišnými farbami. Legenda zobrazuje toto priradenie farieb kategóriám. V našom prípade je kritická cesta zložená z týchto časových kategórií, zhora nadol:

- **Serial impact time** (oranžovou) a **Serial cruise time** (svetlo oranžovou) ukazuje čas strávený sekvenčne vykonávaným kódom.
- **Fully parallel impact time** (zelenou) ukazuje časové kvantum, počas ktorého sú efektívne využité všetky dostupné procesory.
- **Oversubscribed impact time** (modrou) ukazuje množstvo času, počas ktorého boli využité všetky procesory, vláken však bolo zbytočne veľa.

V ideálnom prípade, na viacprocesorovom systéme, by mala časovému rozloženiu dominovať **Fully parallel** časová kategória (všetky zelené farby). Všetky ostatné farby indikujú možnosť vylepšenia výkonu aplikácie. V našom prípade by graf mohol obsahovať viac **Fully parallel time** než **Oversubscribed time**, takže máme priestor na vylepšovanie.

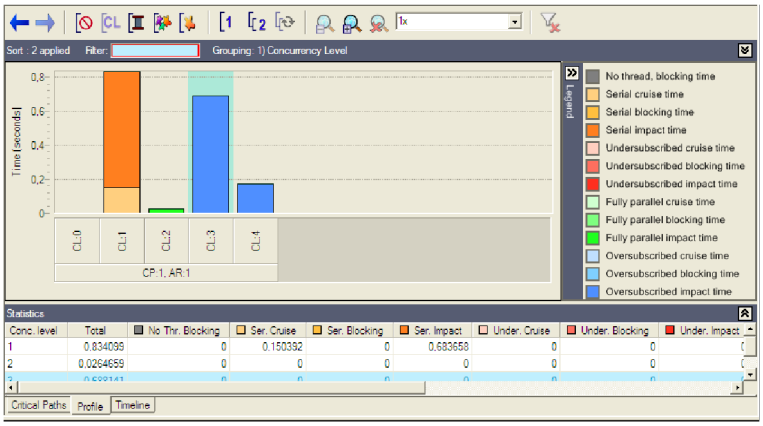
Významy jednotlivých časových kategórií:

1. S kurzorom myši nad grafom kritickej cesty stlačte **F1**, aby sa Vám zobrazil pomocník. Vo vzťahu k aktuálnemu pohľadu sa otvorí téma opisujúca **Critical Paths** pohľad.
2. Kliknite na záložku **Search**, zobrazí sa vyhľadávacia karta. Vpíšte názov časovej kategórie, napr. **Serial cruise time**, a kliknite na **List Topics** pre začatie vyhľadávania. Pomocník prehľadá všetok obsah a zobrazí súvisiace témy.
3. Dvojitým kliknutím na nájdenú tému zobrazíte jej obsah v pravej časti okna pomocníka.
4. Prečítajte si nájdenú tému a prezrite si aj odkazované témy. Napríklad kliknite na odkaz **Dealing with Cruise Time**, aby ste získali užitočné rady ohľadom postupov na zvýšenie efektivity paralelizmu v programe.
5. Kliknite na tlačidlo **Locate** v nástrojovej lište pomocníka, získate tak prehľad o súvisiacich témach na karte **Contents**.

6. Kliknutím na príslušnú tému na karte **Contents** tému zobrazíte.

Profile pohľad

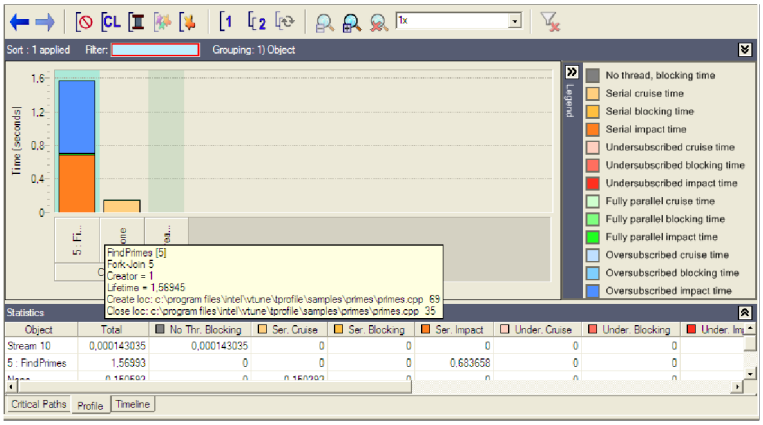
Pohľad **Profile** poskytuje súhrnný prehľad spotrebovaného času na kritickej ceste rozdeleného do časových kategórií. Kliknite na záložku **Profile** pre zobrazenie pohľadu **Profile**:



Štandardne pohľad **Profile** zobrazuje výsledky zoskupené podľa **Concurrency Level**, skrátene **CL**. *Stupeň súbežnosti* značí počet aktívnych vlákien vykonávaných v rovnakom čase pozdĺž kritickej cesty. Zahŕňa práve vykonávané vlákna, ako aj vlákna pripravené na vykonávanie, ktoré nie sú blokované definovaným čakaním alebo blokujúcim príkazom.

Dáta je možné zoskupiť aj podľa **Objects** pre porovnanie časovej závislosti od rôznych softvérových objektov.

Kliknite na ikonu **Set current grouping to Objects** v nástrojovej lište pohľadu pre prepnutie zoskupovania podľa softvérových objektov:



Thread Profiler zobrazí stĺpce prislúchajúce objektom spôsobujúcich vyťaženie na kritickej ceste. Väčšina času kritickej cesty je pridelená objektu typu **Fork-Join** priradeného vláknu `2: FindPrimes`. Názov vlákna je odvodený od názvu funkcie, ktorá je použitá ako vstupný bod vlákna. Za povšimnutie stojí zeleno-šedý priesvitný obdĺžnik v treťom stĺpci, ktorý značí celkové trvanie existencie objektu.

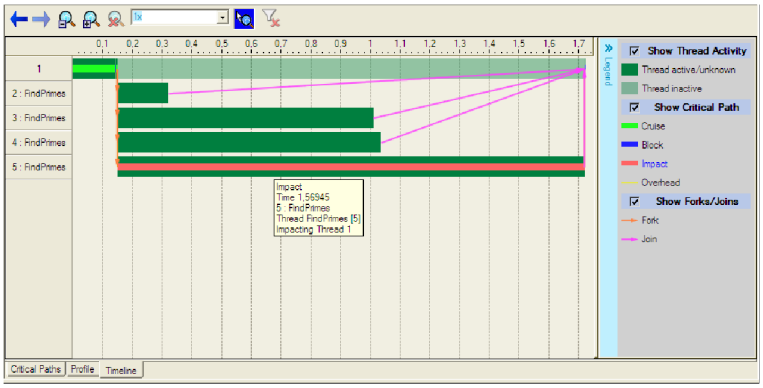
V pohľade **Profile** je ďalej možné:

- Kliknúť na ľubovoľný stĺpec pre zobrazenie podrobných informácií v tabuľke **Statistics** pod grafom (ako je vidieť na obrázku).
- Kliknúť pravým tlačidlom myši na stĺpec pre nastavenie parametrov **Filter** alebo **Group** na určitý typ objektov alebo pre zobrazenie súvisiacich **Source Locations**.

Timeline pohľad

Pohľady **Critical Paths** a **Profile** ukazujú informácie týkajúce sa kritickej cesty. Oproti tomu však pohľad **Timeline** zobrazuje príspevky jednotlivých vlákien programu, či sa nachádzajú na kritickej ceste alebo nie.

Kliknite na **Timeline** záložku pre prepnutie na kartu **Timeline** pohľadu:



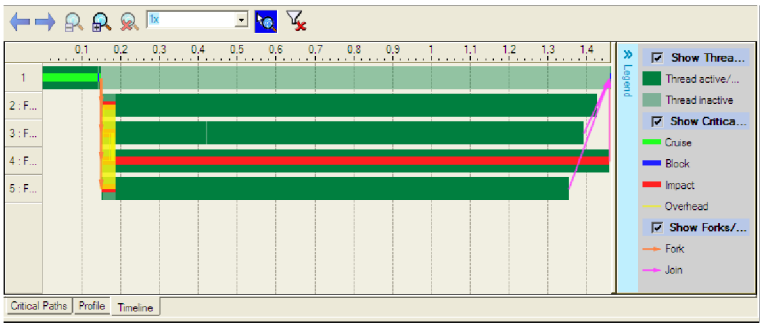
Všimnite si, že všetkým štyrom vytvoreným vláknam trvá postupne vždy viac a viac času dokončenie úlohy. Prvé vlákno, reprezentované obdĺžnikom umiestneným najvyššie, pracuje s množinou malých čísel, čo zaberie málo času. Druhé vlákno spracúva väčšie čísla, trvá mu to dlhšie. Najviac času na splnenie úlohy potrebuje vlákno tretie a štvrté, ktoré pracujú s ešte väčšími číslami. Takže až program dokončí prácu, rozloženie vyťaženia je viditeľne nerovnomerné.

Riešenie: zvýšenie času súbežného vykonávania

Prepísaný kód v príklade `PrimesBalanced.cpp` prekladá (strieda) čísla, ktoré spracúvajú jednotlivé vlákna, takže každé vlákno pracuje rovnako na malých ako aj veľkých číslach. Pre overenie skutočného odstránenia problému nerovnomerného rozloženia záťaže, ktorý sme našli, vykonajte nasledujúce kroky:

1. Preložte opravený kód.
2. Nazhromaždite dáta pomocou Intel Thread Profiler. Je možné vytvoriť novú **Activity** pre upravený súbor `Primes.exe` stlačením klávesy **F5** alebo kliknutím na ikonu **Run Activity** na nástrojovej lište aplikácie.
3. Vyhodnoťte výsledky.

Ak teraz nahliadnete na kartu pohľadu **Critical Paths**, mali by ste vidieť predĺženie modrej oblasti. Pribudol tiež malý kúsok časovej kategórie **Overhead** (žltou), ktorý je



spôsobený čakaním na uvoľnenie kritickej sekcie medzi vláknami. V pohľade **Timeline** je vidieť zlepšenie oproti predchádzajúcemu príkladu:

Modrá oblasť v pohľade **Critical Paths** znamená **Oversubscribed impact time** a je dôsledkom zbytočného použitia veľkého počtu vlákien, ktoré nie sú schopné sa súčasne vykonávať pre nedostatok hardvérových vlákien. Štyri vlákna však predstavujú lepší príklad paralelizmu. Ak znížime počet vlákien v kóde, ktorý je pevne daný makrodefiníciou, modrá oblasť sa bude nahradená zelenou. Tá indikuje efektívne využitie všetkých vlákien v aplikácii.

Intel Thread Checker

Preklad ukážkového kódu

Inštalácia aplikácie Intel Thread Checker obsahuje niekoľko ukážkových programov, ktoré sa vo forme zdrojových kódov nachádzajú v inštalačnom adresári. Ako prvý príklad sa tu nachádza implementácia generátora prvočísel (`Primes`), ktorý používa štandardné API pre vlákna operačného systému Windows. Program hľadá a zaznamenáva prvočísla od jednotky až po 1000 testovaním deliteľnosti nových čísel bezozvyšku predchádzajúcimi prvočíslami. Program pre tento účel vytvorí štyri vlákna. Keďže všetky vlákna pristupujú k jednej premennej, dochádza k potenciálnym dátovým konfliktom. Vo výsledku sa preto môže počet nájdených prvočísel líšiť. Nasledujúci postup ukazuje, ako sa dá použiť aplikácia Intel Thread Checker na nájdenie a odstránenie takýchto chýb.

Preklad kódu:

Postup je rovnaký ako pri preklade ukážkového programu pre aplikáciu Intel Thread Profiler s tým rozdielom, že zdrojové kódy a súbor projektu sa nachádzajú v podadresári inštalačného adresára aplikácie Intel Thread Checker.

Po preložení a niekoľkonásobnom spustení programu môžeme vidieť výstup ako na nasledujúcom obrázku:

Vidíme, že rôzne spustenia dávajú nekonzistentné výstupy. Správnym výsledkom je `"Found 168 primes"`. V tomto prípade je dosť jednoduché odhaliť chybu. Vo veľkých programoch môže byť nekonzistencia vlákien identifikovateľná len veľmi ťažko. V nasledujúcich odstavcoch sa pozrieme na spôsob ako pomocou Intel Thread Checker aplikácie nazbierať dáta a lokalizovať podobné chyby.

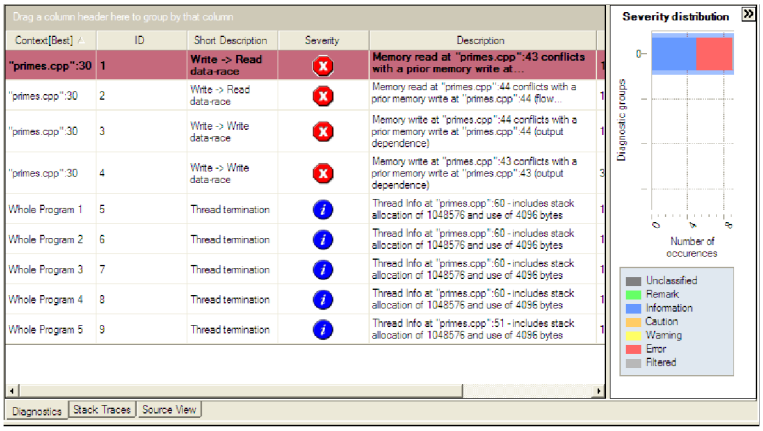
```
$ ./Primes.exe
Determining primes from 1 - 1000
Found 165 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 167 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 168 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 168 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 102 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 168 primes
$ ./Primes.exe
Determining primes from 1 - 1000
Found 168 primes
$ _
```

Zbieranie dát

Vytvorenie projektu v aplikácii Intel Thread Checker je veľmi podobné postupu ako v prípade programu Thread Profiler. Len je potrebné vybrať sprievodcu **Thread Checker Wizard** namiesto **Thread Profiler Wizard**. Po dokončení sprievodcu by sme mali mať spustenú **Activity**, ktorá nazhromaždí dáta pre analýzu.

Analýza výsledkov a oprava kódu

Po dokončení sprievodcu by sa mali zobraziť výsledky ako na nasledujúcom obrázku:



Teraz by sme mali byť schopní analyzovať diagnostické dáta a nájsť nekonzistencie vlákien.

Analýza diagnóz

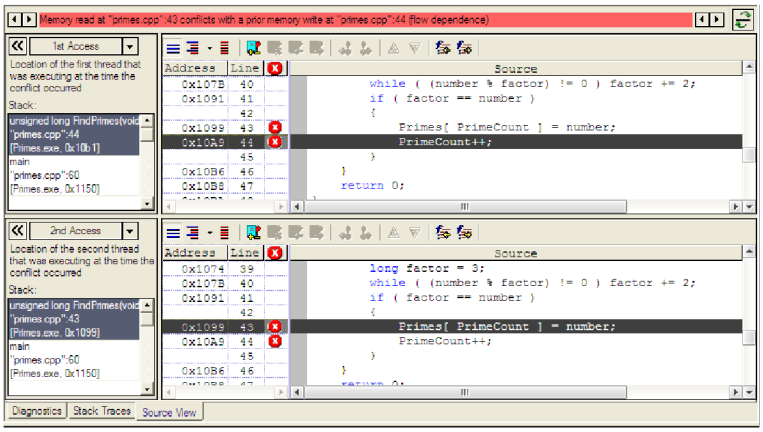
1. Pozrite sa na prvú diagnózu v zozname zvýraznenú červenou farbou s priradeným identifikátorom **ID 1**. Thread Checker identifikoval chybu **Write → Read data-race**. Táto chyba, označená červenou ikonou, je spôsobená nesynchronizovaným prístupom k zdieľanej premennej.

2. Pravým kliknutím myši na prvý riadok a výberom **Diagnostic Help** otvorte pomocníka. Zobrazí sa podrobný popis chyby, možné príčiny jej vzniku a návod na jej odstránenie.

Write → Read dátový konflikt vzniká, keď jedno vlákno zapisuje do zdieľaného pamäťového miesta, kým druhé vlákno sa pokúša to isté pamäťové miesto načítať.

Nájdenie miesta chyby v kóde

1. Kliknite na záložku **Source View**. Karta **1st Access** zobrazuje umiestnenie v kóde, kde sa prvé vlákno pokúšalo zapísať na zdieľané pamäťové miesto. Karta **2nd Access** ukazuje riadok zdrojového kódu, na ktorom došlo k nesynchronizovanému čítaniu zdieľaného pamäťového miesta.



2. Lepší pohľad do zdrojového kódu je možné získať zväčšením príslušnej karty.

Vyriešenie tohto dátového konfliktu sa dá dosiahnuť pridaním kritickej sekcie do kódu aplikácie. Použitím synchronizačného objektu, ktoré zoserializuje čítanie a zápis na riadkoch 43 a 44, je možné potlačiť vedľajší efekt časového multiplexu súbežného vykonávania vlákien. Pre overenie odstránenia chyby sa zopakuje postup s prekladom a podobne ako pri Thread Profiler sa opäť spustí **Activity**. Príklad s vyriešeným problémom sa nachádza aj v adresári ukážkového programu.