Technická univerzita v Liberci

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 Elektrotechnika a informatika
Studijní obor: 1802R022 Informatika a logistika

Použití paralelismu a distribuovaného zpracování dat v praxi

Use of paralelism and distributed processing of data in practice

Zpráva bakalářské práce

Autor: Petr Švub

Vedoucí: Ing. Dalibor Frydrych, Ph.D.

Liberec, Květen 2007

Zadání

Název projektu: Použití paralelismu a distribuovaného zpracování dat v praxi

Řešitel: Petr Švub

Vedoucí učitel: Ing. Dalibor Frydrych, Ph.D.

Zadání:

1. Seznamte se se základy objektového návrhu numerických modelů

2. Seznamte se s technologií vícevláknového zpracování dat v jazyce JAVA

3. Seznamte se se serializací objektů a jejich distribucí pomocí technologie RMI na jiné výpočetní systémy

4. Vytvořte funkční numerický model využívající paralelizační technologie a otestujte jeho vlastnosti

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé BP a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářskou práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářskou práce.

Datum

Podpis

Obsah

Seznam tabulek

Abstrakt/Abstract

Abstrakt

Bakalářská práce je zaměřena na problematiku paralelního zpracování dat a možného využití těchto principů v praxi

V práci jsou rozebrány principy paralelismu, Amdahlův zákon, standard MPI, ale především je soustředěna na paralelizační schopnosti programovacího jazyka Java. Ty jsou zastoupené v technologiích Multithreading, Serializace a Remote Method Invocation.

Na závěr práce je funkční systém, který je schopný distribuovaných výpočtů, otestován na modelové úloze.

Klíčová slova: paralelismus, multithreading, serializace, RMI, distribuované výpočty.

Abstract

This bachelor diploma is focused on problematics of parallel data processing and possible usage of this principles in practice.

In the work, there are explained principles of parallelism, Amdahl's law, MPI standard, but above all is this work focused on parallel abilities of Java programming language. These are supplied by technologies Multithreading, Serialization and Remote Method Invocation.

In the end of this work is functional system, able to carry on distributed computing, tested on model excersise.

Keywords: parallelism, multithreading, serialization, RMI, distributed computing.

Úvod

Na světě existuje mnoho úloh, které jsou velmi náročné na výpočetní techniku, modely předpovědi počasí, simulace vzniku Země nebo dokonce celého vesmíru (Velký Třesk), a lidé stále přidávají další. Tyto úlohy jsou pevně dané, vědci většinou vědí co s nimi mají dělat, nicméně jim k provedení chybí prostředky. Řešení některých úloh by se současnými výpočetními možnostmi trvalo celá léta. Přitom se ve vývoji stále nových a silnějších procesorů blížíme k samotným fyzikálním hranicím. Procesorový čas superpočítačů je navíc velmi drahý, což je také dobrý důvod k hledání jiného způsobu.

Rozdělit úlohu na části a ty pak zpracovávat zároveň na několika výpočetních zdrojích, je myšlenka poměrně jednoduchá. Pro danou úlohu musí ale existovat vhodné rozdělení na části, jež by bylo možné zpracovávat každou zvlášť. Celou touto problematikou se zabývá paralelismus.

Paralelizačních technologií bylo v průběhu času vyvinuto několik. Způsob, jakým je k paralelismu přistupováno závisí především na použitém programovacím jazyce. V případě programovacího jazyka Java jsou to tři technologie. Multivláknové zpracování procesů lze použít především na strojích a vícejádrovými procesory, kde jsou vlákna automaticky rozdělena mezi jednotlivá jádra. Serializace umožňuje objekt, se kterým pracujeme, zabalit a odeslat na jiný výpočetní zdroj, kde je po rozbalení možné pokračovat v práci. A konečně RMI, jež v podstatě zastřešuje serializaci, čímž umožňuje vytvořit kompaktní systém schopný distribuovat výpočet na jiný, třeba i velmi vzdálený výpočetní zdroj. Sladěním těchto tří technologií lze vytvořit robustní systém, kterým je možné modelovou úlohu rozdělit na dílčí výpočty, a ty rozeslat například na výpočetní klastr.

1 Paralelismus

Je známým faktem, že technologie současných počítačů se blíží ke svým fyzikálním hranicím. Dosavadní, poměrně rychlý růst, byl dán dvěma faktory:

1. Možností umístit na jeden čip stále větší množství tranzistorů.

2. Novými poznatky a technologiemi použitými při návrhu vnitřní struktury čipů.

Tento vývoj však nemůže pokračovat do nekonečna. Zcela zákonitě se jednou musíme dostat do míst, kde již začínáme být omezování základními fyzikálními zákony. Jedná se zejména o omezení při minimalizaci součástí čipu, kde již začínáme být limitování velikostí atomů použitých materiálů a omezení rychlosti šíření informací. Toto poslední omezení je známo pod názvem "Argument rychlosti světla" (Speed-Of-Ligt Argument).

Rychlost světla je zhruba 30cm/ns. Signály cestující komunikačním vodičem cestují zlomkem této rychlosti. Je-li velikost čipů 3cm, můžeme říci, že výsledek výpočetní operace nesený signálem na druhou stranu čipu nemůže být proveden vícekrát než 10^{10}krát za sekundu. Redukcí vzdáleností o násobky 10, či dokonce 100, zvýší množství provedených operací zase pouze o tyto násobky; čili dostaneme se na hodnoty kolem 10^{12} operací za sekundu. Navíc, abychom byli schopní dosáhnout těchto hodnot, museli bychom zajistit, že signál bude přenesen v rámci jednoho instrukčního cyklu, což není díky vnitřní architektuře procesorů vůbec jednoduché. Z těchto důvodů začíná být zřejmé, že se již blížíme k hodnotě, kdy začínáme být omezováni maximální možnou rychlostí šíření signálu a dalšími základními fyzikálními principy

Nepřetržitě se pracuje na vývoji nových technologií, jako jsou počítače kvantové, chemické, či dokonce biologické, nicméně než budou tyto technologie uvedeny v praxi, nějaký čas to ještě potrvá. Je zde ovšem technologie, kterou lze výpočetní výkon počítačů razantně zvýšit a kterou máme k dispozici, paralelismus. Rozsáhlá paralelizace může být jednou z cest, jak zajistit další růst výpočetního výkonu hned teď. Použít lze dnešní běžné procesory, které jsou relativně levné, a spolu s moderními síťovými technologiemi vytvořit vhodné prostředí pro distribuované výpočty. Samozřejmě i v tomto případě budeme limitování rychlostí šíření signálu, v porovnání s navýšením výpočetního výkonu je však toto omezení zanedbatelné. Motivaci pro využití paralelních výpočetních systémů můžeme tedy shrnout do tří bodů:

1. Zvýšení rychlosti řešení daného problému. To je důležité zejména v případech, kdy má aplikace stanoveny meze, které musí být přesně dodrženy.

2. Zvýšení propustnosti, tj. možnost řešit více instancí daného problému v jednom okamžiku.

3. Zvýšení výpočetního výkonu při řešení rozsáhlých problémů (simulací atd.)

Myšlenka paralelismu je známá lidstvu již odnepaměti. Například Egyptské pyramidy by nikdy nestály, nebýt naplánování stavby a následné realizace několika etap zároveň pomocí tisíců otroků (jeden, ať sebesilnější, by nikdy nebyl schopen zastat tolik lidí pracujících v kooperaci). Ani v informačních technologiích není paralelizace žádnou novinkou. Například stroj ILLIAC IV, který byl dokončen roku 1976 po více než desetiletém vývoji, měl až 256 procesorů.

1.1 Úrovně paralelizace

Principy paralelismu můžeme do světa počítačů zavádět v několika rovinách. Tou nejnižší (fine grained) je paralelizace pomocí velmi malých, až elementárních podprogramů, které mohou běžet najednou. Tento stupeň se často řeší už na úrovni hardwaru a strojových instrukcí. Dnešní procesory své instrukce zpracovávají zřetězeně (jinými slovy v řetězu, v rourách, pipelined) tak, že jedna část je předzpracovává, několik dalších jednotek souběžně vykonává a další se stará o celkovou synchronizaci. Například: pokud často pracujeme s grafikou, koupíme si grafický akcelerátor a tak ulehčíme samotnému procesoru, který se místo vypočítávání obrazu může zabývat jinými úkoly.

Vyšším stupněm (middle grained) je paralelizace na úrovni procesorů, kde už je zapotřebí invence samotných programátorů vyvíjejících aplikace. Výpočet se musí rozdělit do několika spolupracujících úloh tak, aby mohly běžet pokud možno současně na několika procesorech.

Nejvyšší (coarse grained) úrovní paralelizace se rozumí architektura s řadou procesorů, na kterých simultánně běží několik vzájemně nezávislých úloh. Díky násobnosti procesorů se propustnost systému zvýší, teoretická doba běhu algoritmů se však nezkrátí. Jednotlivé programy se píšou pořád stejně sekvenčně (jakoby pro jeden procesor), jestliže však takových úloh běží v systému více, jejich zpracování se rozdělí na všechny dostupné procesory, v konečném důsledku se docílí výrazného zrychlení.

Nejnižší vrstvu - využití principu paralelnosti na úrovni hardwaru - lze vysvětlit na popisu architektury procesoru Intel P6.

Procesory rodiny P6 byly představeny firmou Intel roku 1995. Prvním zástupcem této skupiny byl procesor Pentium Pro. Při vývoji procesorů rodiny P6 bylo jedním z prvořadých cílů významně překročit výkon procesoru Pentium s použitím stejné výrobní technologie, a to znamenalo, že zvýšení výkonu bylo možné docílit pouze vylepšením mikroarchitektury. Jsou třícestně (three-way) superskalární, zřetězené (pipelined) architektury. Termín "třícestně superskalární" stručně znamená, že použitím paralelních technik bylo docíleno toho, aby byl procesor schopen dekódovat, připravit a kompletně provést průměrně tři instrukce za jeden hodinový cyklus. Aby bylo dosaženo takové propustnosti, procesory rodiny P6 jsou tvořeny dvanáctistupňovým zřetězením (12-stage superpipeline), které podporuje tzv. out-of-order vykonávání, tedy provádění instrukcí v pořadí, které nemusí nutně odpovídat tomu, jak instrukce do procesoru dorazily (integrita dat musí pochopitelně zůstat zachována).

Bus Interface Unit - instrukce a data jsou do procesoru dodávány prostřednictvím jednotky rozhraní sběrnice. Tato jednotka zprostředkovává styk procesoru se systémovou sběrnicí. Pro překlenutí rozdílu mezi rychlým procesorem a relativně pomalou sběrnicí používá BIU navíc tzv. L2 cache paměť pro uchování údajů, u kterých je pravděpodobné, že je bude brzy potřebovat.

Procesor používá ještě jednu cache paměť – L1. Ta je již rozdělena na dvě části. V první části se uchovávají instrukce a ve druhé data.

Fetch/Decode Unit - jednotka volání a dekódování instrukcí má na starosti předzpracování instrukcí. Tato část procesoru čte z L1 instrukční cache paměti proud instrukcí a dekóduje je na tzv. mikroinstrukce. Mikroinstrukce jsou potom poslány do banky dekódovaných instrukcí (Instruction Pool), kde čekají na zpracování dalšími jednotkami procesoru.

Abychom pochopili rozdíl mezi instrukcí a mikroinstrukcí, můžeme si instrukci na chvíli představit jako kartičku s názvem nějaké operace. Mnoho instrukcí je samo o sobě elementárních a potom odpovídají právě jedné mikroinstrukci. Některé instrukce však pojmenovávají natolik složité operace, že procesor není schopen je provést v jednom jediném kroku. Potom se taková instrukce dekóduje na celou sadu mikroinstukcí, které v souhrnu provádějí požadovanou operaci.

Dekodér instrukcí tedy čte postupně jednu kartičku s názvem operace za druhou a podle svých tabulek dává na svůj výstup proud menších kartiček (mikroinstrukcí), které již představují nejjednodušší možné operace procesoru.

Uvnitř jednotky volání a dekódování instrukcí jsou ještě zabudovány různé další složité mechanismy, které se snaží zpracování instrukcí optimalizovat. Asi nejzajímavějším nástrojem je předpovídání větvení programu. Jednotka se dívá hluboko dopředu na instrukce, které následují za právě

zpracovávanou, a už dopředu se je snaží dekódovat. Problémem však jsou různé příkazy větvení, které realizují různé skoky do jiných částí programu. Mechanismus předpovídání větvení se snaží uhodnout, kterou cestou se zpracovávání programu bude ubírat, a zrovna tyto instrukce nabídnout dekodéru k předzpracování.

Samotný dekodér se skládá ze tří paralelních částí: dvou dekodérů jednoduchých instrukcí (Simple-instruction Decoders) a jednoho dekodéru složitých instrukcí (Complex Instruction Decoder). Každý dekodér zkonvertuje instrukci architektury Intel na jednu nebo více mikroinstrukcí. V každém taktu procesoru tak může dekodér vygenerovat až šest mikroinstrukcí - jednu každým ze dvou dekodérů jednoduchých instrukcí a až čtyři dekodérem složitých instrukcí.

Instruction Pool - dekódované mikroinstrukce putují do již zmíněné banky dekódovaných instrukcí. Instruction Pool je v podstatě sada čtyřiceti speciálních registrů, přičemž do každého z nich se vejde právě jedna mikroinstrukce.

Dispatch/Execute Unit - z banky instrukcí jsou jednotlivé mikroinstrukce čteny jednotkou provádění instrukcí , která je tvořena dvěma celočíselnými jednotkami (Integer Units), dvěma jednotkami pro operace s plovoucí čárkou (Floating - point Units) a jednou jednotkou pro přístup do paměti (Memory Interface Unit) - tím se umožní paralelní provádění až pěti mikroinstrukcí najednou.

Důležitý na vykonávání mikroinstrukcí je fakt, že z banky dekódovaných instrukcí mohou být čteny v jiném pořadí (out-of-order), než v jakém do poolu přišly. Tím se velice zvyšuje propustnost celého systému, protože se mohou provádět mikroinstrukce podle toho, jaké exekuční jednotky jsou volné, a ne nutně podle pořadí jednotlivých mikroinstrukcí. Pochopitelně se tím ale zvyšuje složitost systému, protože si Dispatch/Execute Unit musí dávat pozor na datové závislosti mezi instrukcemi.

Retire Unit - zpracovaná mikroinstrukce se vrací zpátky do banky instrukcí, kde čeká na své definitivní dokončení - promítnutí změn do zbytku systému. Jednotka Retire lineárně prohledává Instruction Pool na vykonané instrukce, které už nemají žádné závislosti na jiných instrukcích či neurčených větveních. Jakmile takovou najde, odevzdá výsledky k promítnutí do stavu systému: uloží je do paměti či do univerzálních registrů procesoru (EAX, EBX, ...). Výsledky se realizují v originálním pořadí (tedy v takovém, v jakém instrukce přicházely do procesoru). Takto zpracované instrukce se nakonec z banky dekódovaných instrukcí odstraní. Architektura procesorů Intel je pochopitelně o mnoho složitější, tento stručný popis by měl však poukázat zejména na paralelnost zpracování, která se v procesoru objevuje na každém kroku. Co všechno se provede v jednom taktu procesorových hodin?

- načtou se až tři nové instrukce z vyrovnávací paměti (L1 cache), dekódují se na mikroinstrukce a uloží se do banky dekódovaných instrukcí (Instruction Pool).
- spustí se provádění až pěti mikroinstrukcí uložených v Instruction Poolu z minulých dekódování.
- instrukce zpracované v předchozím taktu se promítnou do univerzálních registrů a stavu systému.

Přitom nestačí jen takové hrubé zřetězení, jak je zde popsáno. Jednotlivé jednotky se uvnitř dále dělí na menší paralelní části, které jsou často navíc v procesoru násobně. Viz například dekodér, který je v procesoru ztrojený nebo vykonávací jednotka, která se skládá z pěti částí schopných pracovat souběžně.

Přes to všechno se ale nejedná o žádný procesor s paralelní architekturou. Tento procesor je klasický, von Neumannovského typu, přičemž principu paralelismu využívají pouze uvnitř svých výkonných jednotek. Navenek se chovají jako sekvenční systémy.

Podle klasifikace počítačových architektur, nevržené v roce 1966 Michaelem J. Flynnem, patří tento typ strojů do skupiny **SISD** (Single Instruction, Single Data - jedna instrukce na jedna data. Z hlediska paralelismu lze počítačové architektury roztřídit ještě o dalších třech skupin:
SIMD (Single Instruction, Multiple Data - jedna instrukce na více dat), do které náleží vektorové počítače. Ty pomáhají urychlit výpočet tím, že dovolují provádět instrukce na řadě (vektoru) hodnot najednou. Vektorové počítače obvykle obsahují jednu řídící a několik výkonných jednotek, přičemž každá výkonná jednotka je schopna zároveň s ostatními provádět stejnou instrukci na svých datech. Tím, že procesor nepracuje se skalárními veličinami, ale rovnou s vektory, se ve speciálních aplikacích dosahuje výrazného zrychlení. Příklady strojů: Cray, NEC SX-4. Za zmínku stojí, že i obyčejná Pentia mají několik vektorových instrukcí. Jsou jimi instrukce MMX (Multimedia Extensions - Pentium, Pentium II) a SSE (Streaming SIMD Extensions - Pentium III) určené pro zrychlení práce s multimédii, audio/video a ve 3D oblasti.

Zkratkou MMX se souhrnně nazývá 57 instrukcí, které se poprvé objevily u procesorů Pentium a u nástupce procesoru Pentium Pro – u Pentia II. Dovolují paralelně provádět nad více datovými vstupy různé matematické operace vyskytující se typicky v aplikacích pro zpracování zvuku, grafiky a videa.

SSE jsou obdobou instrukcí MMX. Poprvé spatřily světlo světa v procesorech Pentium III. Streaming SIMD Extensions zavádí 70 nových SIMD

instrukcí. Na rozdíl od MMX, které dokáží pracovat jen s celými čísly, instrukce SSE zvládají výpočty s pohyblivou desetinnou čárkou.
Architektura **MISD** (Multiple Instruction, Single Data - více instrukcí na jedna data) není příliš obvyklá, vzhledem k tomu, že proud více instrukcí potřebuje také více dat, na která by mohl být aplikován, aby byl efektivní. Nicméně tento typ lze použít v takzvaném redundantním paralelismu, například v letadlech, která potřebují několik záložních systémů pro případ, že by jeden selhal. Také bylo navrženo několik počítačových architektur, které by mohly MISD využít, ale žádný návrh nedošel do stádia nějakého většího rozšíření.
MIMD (Multiple Instruction, Multiple Data - více instrukcí na více dat) je taková výpočetní architektura, ve které více funkčních jednotek provádí různé operace na různá data. Tento termín se ve Flynnově taxonomii objevil až v roce 1972.

Symetrické multiprocesory, které do této skupiny patří, jsou charakteristické tím, že se skládají z několika (obvykle až desítek) procesorů a sdílené paměti. Vše je propojeno nějakým komunikačním subsystémem (sběrnicí). Každý procesor má svůj vlastní proud instrukcí a dat, někdy mohou mít i trochu své lokální paměti (cache), jednotlivé procesory spolu mohou přes sdílenou paměť komunikovat. Ze zástupců symetrických multiprocesorů lze uvést SGI Power Challenge.

Masivně paralelní počítače se skládají z relativně samostatných uzlů, kterých může být dohromady propojeno až na tisícovky. Každý uzel má svůj procesor a vlastní paměť; na každém běží jeho vlastní kopie operačního systému. Úlohy běžící na procesech spolu komunikují předáváním zpráv po meziuzlových linkách. Komunikace je tady na rozdíl od symetrických multiprocesorů pomalejší, ale zase je možno využít řádově vyššího počtu procesorů.

Klastr (cluster) pracovních stanic je občas označován jako distribuovaný systém. V podstatě jde o řadu počítačů propojených sítí (LAN/WAN) tak, aby na nich mohla běžet paralelní úloha (n * PC + síť + software pro komunikaci mezi procesy). Na klastrech pracovních stanic je kouzelné právě to, že jsou často tvořeny obyčejnými počítači, které jiní používají ke kancelářským pracím. Jedná se tedy o velice levné řešení potřeby superpočítače. Hlavním rozdílem mezi klastrem a obyčejnou lokální sítí je, že klastry kladou veliký důraz na rychlost komunikace - předávání zpráv mezi procesory. A tak zatímco pro obyčejnou síť postačí Ethernet (popř. Fast Ethernet), klastry se staví na bázi vícenásobného Fast/Giga Ethernetu, Myrinetu a podobně. Komunikační subsystém je rovněž narozdíl od standardní sítě izolovaný od vnější síťové infrastruktury. Uzly klastru jsou narozdíl od pracovních stanic počítačové sítě zpravidla (ale není to podmínkou) hardwarově/softwarově identické a jsou vyhrazeny pouze pro spolupráci v rámci klastru. Jednotlivými uzly klastrů

jsou obvykle pouze základní jednotky bez periferií (tj. bez klávesnic, monitorů apod.). Parametry OS jsou optimalizovány pro dávkovou průchodnost na rozdíl od pracovních stanic, kde se preferuje interaktivní přístup. Software klastrů umožňuje jednoznačnou identifikaci procesů v rámci celého klastru. Díky tomu může proces na nějakém uzlu poslat signál druhému procesu, který běží na jiném uzlu. To u pracovních stanic není možné. Klastry mohou plnit různé úlohy. Mohou být sestaveny tak, aby dokázaly vyvažovat zátěž (load balancing), kdy požadavky různých klientů jsou směrovány na jednotlivé uzly podle současného zatížení, nebo aby přinesly vysokou dostupnost (high availability), kdy díky určité redundanci přebírá práci jednoho uzlu v případě výpadku jiný uzel, popřípadě lze klastry specializovat na provádění paralelních výpočtů - pak hovoříme o výpočetních klastrech. Typickým příkladem klastru je program SETI, kterého se účastní počítače celého světa. Každému účastníku je zasláno zadání práce, které spočívá v poměrně malém objemu dat, jejichž zpracování ale zabere poměrně delší čas. Protože je doba zasílání zadání poměrně krátká, a doba zpracování mnohem delší, je čas využit správně na řešení úkolu.

1.2 Teorie paralelismu

V paralelismu je obecně cílem rozdělit jednu celistvou úlohu na části, rozdělit tyto části mezi výpočetní jednotky, a tyto synchronizovat tak, aby bylo dosaženo smysluplných výsledků. Důležité je, aby bylo vůbec možné úlohu rozdělit, a následně co nejlépe a nejrovnoměrněji rozložit zátěž mezi zúčastněné výpočetní jednotky. Některé algoritmy je možné rozdělit velmi snadno, například zjištění prvočísel od jedné do tisíce převedeme na paralelní formu tak, že každému z procesorů, které máme k dispozici, přidělíme řadu čísel úměrnou jeho výkonu, a následně od všech převezmeme pouze výsledky. Algoritmy, třeba pro počítání čísla Pí, které potřebují výsledek předchozí operace k tomu, aby mohli pokračovat v počítání už tak snadno paralelizovat nelze, takové úlohy se někdy nazývají jednoznačně sekvenční. Rozdělení zátěže je také důležité. Pokud dostanou některé procesory snadnější úkol, který vyřeší dříve než ostatní, budou po určitou dobu zbytečně čekat a jejich čas nebude správně využit - jistě je snadnější a rychleji proveditelné nalézt prvočísla v intervalu 1 - 100, než nalézt prvočísla v intervalu 901 - 1000, i když v tomto případě by byl časový rozdíl vzhledem k celkové délce trvání celé operace minimální.

Pro návrh algoritmů použitelných pro paralelní počítače se používá model PRAM Paralel Random-Acces Machine. Neřeší synchronizaci a komunikaci, ale umožňuje programátorovi se zaměřit především na využití souběžnosti. V podstatě je jedná o model SIMD se sdílenou pamětí. Parallel Random

Access Machine je tvořený k identickými procesory. Tyto procesory sdílejí společnou (globální) paměť. Když dva procesory v tomto modelu potřebují komunikovat, provádí to pomocí této paměti. Jeden procesor zapisuje data do sdílené pamětí a jiný procesor či procesory je potom čtou. V základním SIMD modelu nejsou žádná omezení pro více souběžných přístupů do různé části sdílené paměti. To už neplatí u vícenásobného přístupu do téhož místa sdílené paměti. Ta obsahuje paměťové buňky číslované od nuly, jejichž počet není omezen. Do každé buňky je možno uložit libovolné celé číslo. Na začátku výpočtu je v několika (obvykle prvních) z nich uloženo zadání úlohy, obsah ostatních buněk není definován. Na konci výpočtu je pak v několika (opět obvykle prvních) buňkách uložen výsledek výpočtu. Každý z procesorů dále má své vlastní lokální paměťové buňky pro celočíselné hodnoty, které jsou stejně jako buňky globální paměti indexovány čísly od nuly.

Jsou čtyři typy PRAMů s ohledem na to, jak mohou procesory číst nebo zapisovat data do paměťových buněk:

1. Exclusive Read Exclusive Write (EREW) PRAM: Žádným dvěma procesorům není dovoleno READ nebo WRITE do téže buňky sdílené paměti najednou.

2. Exclusive Read Concurrent Write (ERCW) PRAM: Je povolen zápis do buňky několika procesorů současně, číst ale může jenom jeden.

3. Concurrent Read Exclusive Write (CREW) PRAM: Současná čtení téže buňky paměti jsou dovolena, ale pouze 1 procesor se smí pokusit zapsat do dané buňky.

4. Concurrent Read Concurrent Write (CRCW) PRAM: Jsou dovoleny jak současné čtení tak současné zápisy téže paměťové buňky.

Procesory mohou mezi sebou komunikovat a spolupracovat v řešení problému, nebo mohou pracovat nezávisle, často pod dohledem dalšího procesoru, který rozděluje práci a sbírá výsledky.

Komunikace je možná dvěma základními způsoby. Buďto pomocí sdílené paměti (případ modelu PRAM), kdy se jedná o různě velký blok paměti RAM, který vytvoří jeden proces, a kam mohou ostatní procesy přistupovat. Je to poměrně snadno programovatelné, protože procesory sdílejí stejný pohled na data a komunikace mezi nimi může být tak rychlá, jak rychlý je přístup do paměti. Z toho důvodu však také může docházet k zácpám, pokud by se více procesů najednou pokoušelo dostat ke stejné informaci. Proto multiprocesorové počítače se sdílenou pamětí nejsou příliš dobře škálovatelné, většina jich pracuje s maximálně deseti procesory. Komunikace pomocí

předávání zpráv (Message Passing) je založena na posílání volání funkcí (ne posílání samotných funkcí, ale pouze odkazů - volání na ně), signálů nebo paketů dat určitým příjemcům (takový systém využívá i technologie Java RMI a jí podobné). Programovací jazyky založené na předávání zpráv, ho definují jako nesynchronizované posílání objemu dat kopírováním druhému účastníku komunikace. Tento koncept je vyšší úroveň komunikace pomocí jednotlivých paketů, jelikož zasílané zprávy mohou být mnohem větší než paket. Předávání zpráv se může dít pomocí společné sběrnice nebo přes vzájemné propojení sítí různých topologií (strom, kruh, hvězda, hypercube, mesh, a podobně). Paralelní počítače založené na vzájemném propojení sítí musí používat nějaký způsob routingu, aby bylo zajištěno předáváním zpráv i mezi uzly, které nejsou přímo propojeny. Komunikační médium, použité pro komunikaci mezi procesory by mělo být ve velkých multiprocesorových systémech přístupné dle jisté hierarchie.

K tomu, abychom byli schopni hodnotit paralelní algoritmy nebo architektury vzhledem k jejich efektivitě řešení určitého problému, je nutné nejprve zavést příslušnou notaci. K hodnocení paralelních algoritmů lze mimo jiné používat následující kritéria:

p počet procesorů

W(p) celkový počet operací vykonaných p procesory. Také bývá označeno jako výpočetní práce, nebo energie.

T(p) doba provádění operace s p procesory (Execution time)

$$T(1) = W(1)T(p) \leq W(p) \tag{1}$$

S(p) zrychlení (Speed-up)

$$S(p) = \frac{T(1)}{T(p)} \tag{2}$$

E(p) efektivita (Eficiency)

$$E(p) = \frac{T(1)}{pT(p)} = \frac{S(p)}{p} \tag{3}$$

V oboru informačních technologií je slovo efektivita používáno k popisu několika požadovaných vlastností algoritmu, a to především rychlosti (čas, potřebný k dokončení operace) a prostoru (využití paměti algoritmem). Optimalizací se dosahuje maximální možné efektivity, dle okolností se zaměřující

buď na snížení náročnosti na paměť na úkor rychlosti, nebo naopak. Rychlost algoritmu je možno ovlivňovat několika způsoby, například často používaným způsobem je zrychlit algoritmus na úkor místa - případ, kdy je lepší výsledek výpočtu někam uložit, namísto opětného přepočítávání vždy, když je daný výsledek potřeba. Termín prostor algoritmu představuje jednak přeložený spustitelný program na disku, to může být redukováno mechanismy pro vykonávání rozhodnutí za běhu (run-time), jako jsou například virtuální funkce. To je ovšem na úkor rychlosti. Druhá část prostoru, který algoritmus využívá, je dočasná paměť, obsazená během provádění operací. Práce navíc, spojená s paralelní verzí kódu ve srovnání se sekvenční verzí, většinou procesorový čas, a vyšší využívání paměti kvůli synchronizaci, komunikaci a vytváření a rušení paralelního prostředí, je zahrnuta v (paralelních) přídavných nákladech (parallel overhead). Paralelní zrychlení je čas sekvenční operace, děleno časem té samé operace v paralelním provedení. K dosažení maximálního paralelního zrychlení lze použít Amhdalův zákon, podrobně rozebraný níže. Dále je zde pojem škálovatelnost - schopnost paralelního programu efektivně využít rostoucí počet procesorů a dosáhnout tak paralelního zrychlení.

V roce 1967 americký počítačový architekt původem z Norska, Gene Myron Amdahl, formuloval vztahy pro zrychlení výpočtu dosažitelného paralelizací. Tyto vztahy, známé jako Amdahlův zákon, lze použít k výpočtu předpokládaného zrychlení paralelizovaného algoritmu vzhledem k algoritmu neparalelizovanému, a ve své době měli výrazný vliv na rozvoj v oblasti paralelních výpočtů. Amdahlův zákon vychází z předpokladu, že každá aplikace má určitou část, kterou nelze paralelizovat a tudíž je nutné ji provést sekvenčně. Dále v textu budou použity tato označení:

T_c - celkový čas potřebný pro zpracování úlhy na jednom procesoru

T_p - čas potřebný pro zpracování části úlohy, kterou lze rozdělit na nezávislé dílčí úlohy

T_s - čas potřebný pro zpracování části úlohy, kterou je nutné zpracovat sekvenčně

Z definice plyne že $T_c = T_p + T_s$. Předpokládejme, že paralelní část lze rozdělit na k stejně velkých a nezávislých částí a že máme k dispozici alespoň k procesorů. Pak mohou být všechny podúlohy zpracovány současně a celá úloha pak bude zpracována v čase $T(k)$, který lze určit vztahem

$$T(k) = T_s + \frac{T_p}{k}$$

Jedním z nejdůležitějších parametrů, které udávají zbýšení výkonnosti získané paralelizací je zrychlení (vztah 2).

S pomocí Amdahlova zákona lze vypočítat předpokládané zrychlení aplikace po zavedení paralelizace. Například, pokud paralelizovaná implementační část algoritmu může provozovat 14% operací algoritmu paralelně a tím pádem rychleji (zatímco zbylých 86% operací paralelizovat nelze), podle Amdahlova zákona bude maximální možné zrychlení paralelizované verze algoritmu $\frac{1}{1-0.14} = 1.163$. Paralelizovaná verze programu bude tedy 1.163krát rychlejší, než jeho neparalelizovaná verze.

Označme symbolem β podíl paralelizovatelné části úlohy, můžeme vztah pro zrychlení upravit do následující podoby

$$S(k) = \frac{T_c}{T(k)} = \frac{1}{(1-\beta) + \frac{\beta}{k}}, kde \beta = \frac{T_p}{T_c}$$

Jednoduše lze dokázat, že pro zrychlení platí $S(1) = 1$ a $1 \leq S(k) \leq k$. Nicméně to nejdůležitější co lze z výše uvedeného vztahu odvodit je, že ať použijeme sebevětší počet procesorů, nikdy nedosáhneme většího zrychlení než $1/(1-\beta)$. Pokud bude nutné provést například pětinu výpočtu sekvenčně ($\beta = 0.8$), lze dosáhnout zrychlení nejvýše pět.

Další důležitou metrikou určující úspěšnost paralelizace je účinnost. Ta se mění s počtem nasazených procesorů. Dosazením z Amdahlova zákona do vztahu 3 pak pro účinnost dostáváme

$$\varepsilon(k) = \frac{S(k)}{k} = \frac{1}{\beta + (1-\beta)k}$$

Účinnost tedy s počtem použitých procesorů klesá (vyjma krajního případu $\beta = 1$).

Amdahlův zákon je někdy spojován se zákonem snižující se návratnosti. Pokud vezmeme objekt, který chceme vylepšit, postupným vylepšováním zpozorujeme monotónně se snižující zlepšení. V tomto případě je to tak, že každý další přidaný procesor zrychlí výpočet úlohy o něco méně.

Na závěr této kapitoly uvedu ještě citaci Gene Amdahla z roku 1967:

> For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution...The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate

processor…At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome.

(Po více než desetiletí bylo předpovídáno, že uspořádání samostatného počítače již dosáhlo svých hranic, a že skutečně výrazných pokroků lze dosáhnout pouze propojením většího množství takovýchto počítačů tak, aby to umožňovalo kooperativní řešení. Povaha přídavných nákladů v paralelismu (komunikace, koordinace…) se však ukazuje být sekvenční, a proto zřejmě nebude možné ji přizpůsobit paralelním technikám. Tyto přídavné náklady proto pokládají horní hranici výkonnosti na násobek pěti až sedmi běžné sekvenční zpracovací míry, a to i v případě, že budou obstarávány samostatným procesorem. V každém případě je těžké předpovědět, jak budou předchozí překážky v sekvenčních počítačích efektivně překonány.)

2 Technologie využivající paralelismus

2.1 Mulitvláknové zpracování procesů

S příchodem tzv. preemptivního multitaskingu (MS Windows ho používají od verze 95, jádro Unix od svého vzniku), což je technologie umožňující operačnímu systému přidělovat strojový čas střídavě mezi běžící aplikace a tím mezi nimi přepínat, dostali vývojáři aplikací do ruky velmi silný nástroj: tzv. procesy a vlákna. Jedná se o skutečné, pravé nástroje paralelního programování. Systém nemusí přepínat jen mezi aplikacemi (jako celky), ale toto přepínaní, uspávání a buzení se může dít i na nižší úrovni. V takovém případě hovoříme o tzv. vláknech. Vlákny jsou myšleny jednotlivé akce, které aplikace provádějí.

I v dnešní době je stále velké množství programů, které běží jako jednotlivá vlákna, což způsobuje problémy ve chvíli, kdy je třeba ošetřit několik událostí najednou. Například takový program nemůže zároveň vykreslovat obrázky a číst vstup z klávesnice, musí věnovat plnou pozornost vstupu z klávesnice, postrádajíc tak možnost, zvládnout více událostí najednou. Ideální řešení tohoto problému je hladký běh dvou částí programu najednou. To nám právě vlákna umožňují. Díky vláknům dokáží aplikace dělat více činností naráz. Přesněji řečeno - dokáží budit dojem, že dělají více činností naráz, neboť jednoprocesorový počítač může v jednom okamžiku fyzicky provádět jen jednu jedinou činnost. Zdání víceúlohového zpracování je vzbuzeno rychlým přepínáním mezi běžícími aplikacemi, které je prováděno operačním systémem. Systém neustále (velmi rychle) přepíná mezi jednotlivými procesy a jejich vlákny: střídavě uspává běžící úlohu a budí ten spící úkol, který je právě na řadě. Podrobněji to lze vysvětlit na příkladu. Předpokládejme, že aplikace provádí čtyři činnosti. Každá z těchto činností je implementována do jednoho vlákna. Jakmile potom systém má rozhodovat o přidělení strojového času, nebude předávat své systémové prostředky (strojový čas) celé aplikaci jako celku, ale bude spouštět (budit) postupně její jednotlivá vlákna. Vlákna tedy umožní rozčlenit aplikaci na několik podúloh, které se budou ve svém běhu rychle střídat a vzbudí dojem, že běží všechny zároveň.

Z toho plyne jeden velmi důležitý důsledek, jehož existenci je třeba mít neustále na paměti, aby bylo možné předejít případným problémům. Vývojáři aplikací jako takoví totiž nemají žádnou možnost, jak exaktně určit, které vlákno aplikace má zrovna běžet. Vše je plně v kompetenci operačního systému a toto je třeba si plně uvědomit. Nelze tedy třeba počítat s tím, že "vlákno A dělá kratší činnost než vlákno B, a proto skončí svou činnost dříve". Nic takového nelze předpokládat, protože systém nemusí systémový čas přidělovat rovnoměrně.

Různá meziprocesová komunikační příslušenství, jako roury, sdílené paměti, semafory, atd., umožnila vytvořit komplexní aplikace pomocí mnoha spolupracujících vláken. Nicméně, pouze s jedním společným procesorem mohly sice tyto aplikace používat víceprocesové modely, ale rychlost zpracování dat zůstávala stejná. Jako více procesů na jednom počítači, mohou běžet i vlákna zdánlivě paralelně, ale teprve když je spustíme na více-procesorovém stroji stane se tato zdánlivá paralelizace skutečnou. Narozdíl od procesů, vlákna sdílejí stejný adresový prostor, a proto mohou číst stejné proměnné a stejná data. Při psaní více-vláknových programů je třeba dát si velký pozor, aby žádné vlákno nemohlo rušit práci jiného vlákna. Takovou situaci je možné přirovnat ke kanceláři, ve které zaměstnanci pracují současně a zároveň nezávisle, až do chvíle, kdy musí použít některé ze společných prostředků nebo když spolu potřebují komunikovat. Jeden se domluví s druhým jen pokud ten druhý právě poslouchá a pokud mluví oba stejným jazykem. Zaměstnanec také může použít tiskárnu jen pokud ji právě nepoužívá někdo jiný. Programátor musí tedy zajistit, aby byla vlákna koordinována a mohla vzájemně spolupracovat.

Ve více-vláknovém programu je vlákno vybráno z fronty vláken připravených k běhu a následně spuštěno na procesoru který je k dispozici. Operační systém může vláknu procesor odebrat a poté ho vložit do fronty. Druhů front je několik, například již zmíněná fronta pro vlákna připravená k běhu, fronta pro blokovaná vlákna, atd. Také Java Virtual Machine (virtuální stroj Java, JVM) může manipulovat s vlákny - přesouvat je z fronty vláken připravených k běhu na procesor, kde mohou začít vykonávat své instrukce.

Preemptivní multitasking používá k práci s vlákny i jazyk Java. Vláknu je přidělena priorita a podle toho se pak střídají v běhu. Nejdříve je procesor přidělen těm s vyšší prioritou a postupně se na řadu dostávají vlákna méně důležitá. Pokud má několik vláken prioritu stejnou, jejich zpracování závisí na operačním systému, který je pro běh programu používán. Například systém Windows používá metodu dávkování času, což zajišťuje střídání vláken se stejnou prioritou na procesoru v pravidelných intervalech. Vlákno může být v jednom ze čtyř možných stavů: běžící vlákno právě provádí své instrukce, pozastavené vlákno čeká a může být vráceno do stavu běžící, pokud je mu to dovoleno, blokované vlákno čeká na použití nějakého příslušenství, které je právě používáno jiným vláknem, a není k němu možný přístup, a přerušené vlákno bylo ukončeno bez možnosti návratu.

Tvůrci jazyka Java nám dali k dispozici dva způsoby vytvoření vláken: implementace rozhraní a rozšíření (extends) třídy. Rozšíření třídy je způsob, jakým Java dědí metody a proměnné od rodičovské třídy (třída, která je rozšiřována). V prostředí jazyka Java je ale možné podědit pouze od jedné třídy. Toto omezení může být překonáno implementací rozhraní, což je asi nejob-

vyklejší způsob tvorby vláken. Rozhraní umožňuje programátorovi vytvořit podklad pro tvorbu třídy. Jsou používána pro rozvržení požadavků pro třídy, ve kterých je pak zajištěna implementace. Rozhraní vše nastaví a připraví a třída nebo třídy, které rozhraní implementují se postarají o veškerou práci.

Mezi rozhraním a třídou je několik podstatných rozdílů. Zaprvé, rozhraní může obsahovat pouze abstraktní metody a proměnné typu `static final`, což jsou vlastně konstanty. Třídy obsahují implementaci metod, a mohou obsahovat všechny druhy proměnných, včetně konstant. Zadruhé, v rozhraní není tělo žádné metody, třída, která rozhraní implementuje, musí obsahovat implementaci všech metod, deklarovaných v rozhraní. Rozhraní má schopnost dědit od jiných, a na rozdíl od třídy, i více rozhraní. V programu nelze vytvořit instanci rozhraní.

Prvním způsobem tvorby vlákna je tedy jednoduše rozšířit třídu `Thread`. Tento způsob je možný pouze pokud není třeba rozšířit ještě nějakou jinou třídu, jak bylo vysvětleno výše. Třída `Thread` je definována v balíčku `java.lang`, který je třeba importovat.

Druhý způsob, implementace rozhraní je flexibilnější v tom smyslu, že třída bude moci v případě potřeby stále ještě rozšířit jinou třídu. Rozhraní `Runnable` obsahuje pouze jedinou abstraktní metodu `public void run` a žádné konstanty, je tedy velice krátké. Stejně jako třída `Thread` se nachází v balíčku `java.lang`.

2.2 Serializace objektů

Krátce po vzniku prvních jazyků pro objektově orientované programování začali jejich vývojáři cítit potřebu persistence nebo také stálosti jejich objektů, čímž je míněno, že životní cyklus objektu není omezen na program, který ho vytvořil. Obvykle to znamená schopnost uložit jej do sekvenčního souboru a potom znovu načíst nebo třeba i převádět mezi propojenými počítači. Jazyk Java tento problém řeší pomocí rozhraní `Serializable`. Toto rozhraní bylo do Javy přidáno ve verzi 1.1 za účelem podpory nových vymožeností, především objektového modelu Java Beans a technologie RMI (která přenáší serializované objekty po síti - viz kapitola 2.3). Je to poměrně jednoduchý, ale velmi účinný způsob serializace objektů, který převede objekt na posloupnost bytů tak, aby mohl být později znovu obnoven. Tato technika funguje i v síťovém prostředí - je možné serializovat objekt na počítači s operačním systémem Windows a poté ho poslat po síti na počítač s operačním systémem Unix, kde bude objekt znovu bez problémů obnoven. Navíc serializace může být použita na objekt téměř jakékoliv systémové třídy a také na jakýkoli objekt, který programátor vytvoří, protože po rozšíření jazyka o možnost serializace bylo upraveno mnoho tříd standardních knihoven, včetně všech

objektových reprezentací primitivních datových typů, všech kontejnerových tříd a řady dalších.

Za účelem serializace je nutné serializovanému objektu zajistit implementaci rozhraní `Serializable`. Nejprve vytvoříme objekt typu `OutputStream`, který následně zabalíme do objektu `ObjectOutputStream`. Dalším krokem je volání metody `writeObject()`, která náš objekt převede na posloupnost bytů a odešle ho do výstupního proudu typu `OutputStream`. Pokud chceme objekt znovu vyvolat, zapouzdříme instanci třídy `InputStream` do objektu `ObjectInputStream` a zavoláme metodu `readObject()`. Takto získaný odkaz na objekt typu `Object`, musíme následně převést na správný typ (viz tabulka 1).

Pro serializaci objektů mezi systémy musí být oba tyto systémy obeznámeny se třídou transportovaného objektu a oba také musí mít k dispozici stejný soubor `*.class`, ze kterého objekt pochází. Nejobvyklejší použití serializace je přenos objektu mezi klientem a serverem, systémy jsou zde tedy myšleny dva vzájemně vzdálené systémy propojené sítí. Nicméně může nastat situace, kdy se jedná pouze o jeden systém, a persistentní objekt je rekonstruován do verze programu odlišné od té, která ho serializovala. V tom případě deserializace selže, pokud byla třída objektu změněna za určitou mez. Pokud není splněna první podmínka, bude vrácena vyjímka `ClassNotFoundException`, pokud není splněna druhá podmínka, vrátí kompilátor výjimku `InvalidClassException` (místní třída není kompatibilní).

Pokud je objekt serializován, je serializováno všechno uvnitř objektu, včetně jiných serializovatelných objektů, na které ukazují odkazy. Tento příjemný rys serializace ušetřuje spoustu práce a také zabraňuje přemíře chyb při kompilaci. Pokud některý z odkazovaných objektů neimplementuje rozhraní `Serializable`, kompilátor vrátí výjimku `NotSerializableException` a stačí chybějící rozhraní doplnit. Je to mnohem jednodušší, než muset testovat rekonstruovaný objekt pro potvrzení správného průběhu serializace.

Kdy tedy můžeme serializaci použít? Důležité je hlavně zvážit, jak užitečný pro nás bude objekt po tom, co bude obnoven. Například běžící vlákno je špatný kandidát pro serializaci. Objekty, které implementují námi napsané metody mohou mít také problémy, protože stav těchto metod není součástí serializačního procesu. Ani objekty, které mají přímé napojení na soubory, sockety nebo na jiné vstupní a výstupní toky nejsou serializovatelné, protože jejich funkčnost je vázána na systémové zdroje, které nebudou na jiném, vzdáleném systému k dispozoci, ani není ničím zajištěno že po deserializaci budou stále tam kde byly před serializací. Většina těchto případů je zřejmá, ale je užitečné vědět, že serializace má své hranice. Pokud si nepřejeme aby mechanismus serializace automaticky ukládal a obnovoval určité podobjekty, například proto, že obsahují určité informace, které by se v žádném případě

neměly dostat do nepovolaných rukou, použijeme klíčové slovo `transient`. Říkáme tím, že o uložení nebo obnovení daného podobjektu chceme rozhodovat sami. Po obnovení celého objektu mají tyto podobjekty hodnotu null. Další možností je implementace rozhraní `Externizable`. U objektu, který toto rozhraní implementuje není automaticky serializováno nic. Jestli ale přesto serializovat, uvedeme ji v metodě `writeExternal()`.

2.3 Technologie RMI

Distribuovaný systém je program nebo sada programů, které pro svůj běh využívají více než jeden výpočetní zdroj. Metody distribuovaných výpočtů pokrývají široké spektrum, od více-vláknových aplikací, přes aplikace, které pro svůj běh využívají jeden systém (například síťový klient a server na jednom počítači), až po aplikace, kde klientský program a server běží na počítačích často velmi vzdálených (webové aplikace).

Systémy pro distribuované výpočty existovaly už před vznikem jazyka Java. Tradiční mechanismy jsou RPC (Remote Procedure Call - vzdálené volání procedur) a CORBA (Common Object Request Broker Architecture). Java přidává technologii RMI (Remote Method Invocation - vzdálené volání metod), která je v podstatě objektově orientovaný RPC mechanismus. Technologie RMI byla poprvé představena ve verzi JDK 1.1 (Java Development Kit). Zjednodušeně řečeno, vzdálené volání procedur je schopnost používat pro běh kódu jiný, vzdálený počítač, přičemž se celá sestava chová, jako by pro svůj běh používala pouze počítač, ze kterého kód spouštíme. Programátorovi umožňuje vytvářet distribuované systémy založené na jazyce Java, ve kterých mohou být metody vzdálených objektů volány z jiných JVM, třeba i na jiných počítačích. RMI využívá "marshalling" objektů (převedení do proudu dat tak, aby mohl být později opět restaurován), čímž se nezmění informace o jejich typech, a tak podporuje skutečný objektově orientovaný polymorfizmus. Předáváme-li objekt jako parametr, závisí způsob předání na tom, zda je tento objekt vzdálený (tzn. zda exportoval své metody). V případě normálních objektů, je při předávání parametrem nebo jako návratové hodnoty tento objekt poslán přímo. Pokud jde o vzdálený objekt, je vždy vytvořen jeho zástupný objekt, a ten je zaslán namísto původního objektu. RMI, stejně jako CORBA používá pro komunikaci stub (pahýl, zástupný objekt) a skeleton. Stub je lokálním zástupcem vzdáleného objektu v klientově JVM. Klient vyvolává metody na stubu a ten je odpovědný za jejich provedení na vzdáleném objektu. V RMI stub implementuje totožná rozhraní, která implementuje vzdálený objekt jako vzdálená rozhraní. Po vyvolání metody na stubu provede tento postupně: navázání spojení se vzdáleným JVM, který obsahuje vzdálený objekt, marshalling parametrů, po dokončení me-

tody provede unmarshaling vrácené hodnoty a vrátí výsledek klientovi. Skeleton je naopak zástupný objekt v JVM, které obsahuje vzdálený objekt. Po zkontaktování vzdáleného JVM klientem je předáno řízení skeletonu, který má na starosti unmarshalling parametrů, vyvolání metody na příslušném objektu, marshalling vrácené hodnoty a zaslání klientovi.

Existují dva druhy tříd, které mohou být v technologii Java RMI použity. Prvním typem je vzdálená třída, jejíž instance může být užívána vzdáleně, přičemž na objekt takovéto třídy může být odkázáno buď uvnitř adresového prostoru, kde objekt vznikl a ten potom může být používán jako jakýkoliv běžný objekt, nebo uvnitř jiného adresového prostoru. Na takový objekt se přistupuje pomocí identifikátoru objektu. Ve většině případů se pak identifikátory objektů používají stejně jako běžné objekty. Druhým typem třídy je serializovatelná třída, jejíž instance mohou být kopírovány z jednoho adresového prostoru do jiného. Instance serializovatelné třídy se nazývá serializovatelný objekt. Pokud je serializovatelný objekt předáván jako obyčejný parametr vzdáleného volání metod (RMI), potom hodnota objektu bude kopírována z jednoho adresového prostoru do jiného. Naproti tomu, pokud je vzdálený objekt předáván jako parametr, bude kopírován pouze identifikátor objektu.

Jednodušší z obou druhů tříd je serializovatelná třída. Aby se třída stala serializovatelnou, musí implementovat rozhraní `java.io.serializable`, přičemž podtřídy takové třídy jsou rovněž serializovatelné. Za běžných okolností jsou data uvnitř serializovatelné třídy také serializovatelná.

Použití serializovaných objektů ve vzdáleném volání metod je zřejmé. Jednoduše předáme objekt s použitím parametru nebo jako vracenou hodnotu. Typ parametru nebo vracené hodnoty je serializovatelná třída. Klient i server musí mít oba přístup k definicím všech serializovatelných tříd, které jsou používány. Pokud klient a server běží na různých počítačích, definice serializovatelných tříd by měly být zkopírovány z jednoho počítače na druhý.

Definovat vzdálenou třídu je poněkud složitější než definovat serializovatelnou třídu. Vzdálená třída má dvě části: rozhraní, a třídu samotnou. Vzdálené rozhraní musí v definici splňovat několik požadavků: musí být veřejné (public), musí rozšiřovat rozhraní `java.rmi.Remote` a každá metoda v rozhraní musí obsahovat deklaraci `throws java.rmi.RemoteException`. Samotná vzdálená třída musí implementovat toto rozhraní, dále by měla rozšiřovat třídu `java.rmi.server.UnicastRemoteObject`. Objekty takové třídy existují v adresovém prostoru serveru a mohou být vzdáleně volány. Ačkoliv jsou i jiné způsoby jak definovat vzdálenou třídu, toto je nejjednodušší cesta jak zajistit, aby její objekty mohly být užívány jako vzdálené objekty. Vzdálená třída může obsahovat i metody, které nejsou obsaženy ve vzdáleném rozhraní. Tyto metody ale mohou být volány pouze lokálně.

Stejně jako v případě serializovatelné třídy je nezbytné, aby klient i server měly přístup k definicím vzdálené třídy. Server pro svůj běh potřebuje definici jak vzdálené třídy, tak i vzdáleného rozhraní, avšak klient používá pouze vzdálené rozhraní. Vzdálené rozhraní tak představuje typ identifikátoru objektu, zatímco vzdálená třída reprezentuje typ objektu. Pokud je vzdálený objekt používán vzdáleně, jeho typ musí být deklarován jako typ vzdáleného rozhraní, ne typ vzdálené třídy. Program napsaný v Javě musí nastavit bezpečnostního manažera (security manager), který rozhodne o bezpečnostní politice. Bezpečnostní politiku lze nastavit vytvořením objektu `SecurityManager` a voláním metody `setSecurityManager` z třídy `System`. Důležité pro komunikaci mezi serverem a klintem je také nastavení v souboru `.java.policy`. Musí zde být hlavně povoleno připojení na danou adresu s použitím daného portu.

2.4 Standard MPI

Přestože byl model předávání zpráv vždy široce přijímán, až do začátku devadesátých let minulého století existovaly jeho realizace, tj. systémy předávání zpráv, pouze jako proprietární produkty jednotlivých výrobců nebo vývojářských skupin, které pochopitelně byly navzájem nekompatibilní. Přestože některé z těchto systémů získaly značnou popularitu, na prvním místě zde jmenujme PVM , potřeba vytvořit opravdový standard pro rozšiřující se komunitu uživatelů víceprocesorových systémů se stala velmi akutní.

V roce 1992 vznikla standardizační pracovní skupina, která v listopadu téhož roku iniciovala vznik MPI fóra (MPI Forum). Do práce na novém standardu se v něm postupně zapojilo na 175 odborníků ze čtyřicítky významných organizací, zahrnující výrobce paralelních počítačů, vývojáře softwaru, akademické pracovníky či vědce z aplikačních oblastí. O rok později byl na světě první návrh, jenž v dubnu 1994 vyústil v oficiální standard MPI verze 1.0. Práce se však nezastavila, standard byl korigován verzemi 1.1 (1995) a 1.2 (1996). MPI-2 z července 1996 obohatilo standard o desítky nových funkcí.

Standard Message Passing Interface je tedy souhrn specifikací pro předávání zpráv (message passing) navržený pro zvýšení výkonu na masivně paralelních počítačích i na klastrech pracovních stanic a jako takový je první standardizovaný, komerčně nezávislý projekt pro model předávání zpráv. Cílem Message Passing Interface bylo hlavně umožnit přenositelnost paralelních aplikací na úrovni zdrojového kódu a vytvořit předpoklady pro efektivní implementaci modelu předávání zpráv. K dílčím požadavkům náležely podpora heterogenních paralelních struktur, sémantika nezávislá na programovacím jazyku a blízkost k existujícím nástrojům, také jazykové rozhraní pro C/C++ a Fortran bylo jedním z vedlejších předpokladů. Výhody pou-

žívaní této knihovny při vývoji naplňují cíle vývojářů co se týče stability, přenosnosti a flexibility. Standard byl příznivě přijat jak uživateli a programátory, tak i výrobci paralelní výpočetní techniky a dnes je dostupný minimálně ve verzi 1 prakticky na všech víceprocesorových platformách. MPI Není standardem ISO ani IEEE, ale v průběhu času se stala "průmyslovým standardem" pro psaní programů na všech platformách HPC (High Performance Computing). MPI je zástupcem explicitní paralelizace, kdy je plně na programátorovi, aby detekoval paralelismus v algoritmu a implementoval ho prostřednictvím konstruktů MPI v jeho kódu. Nic v tomto směru se neudělá automaticky, nic například nezařídí překladač. Není to knihovna jako taková, ani software, nýbrž forma, předpis, specifikace, jak má software aspirující na název MPI vypadat, z čeho se má skládat a jak se má chovat. Implementace, které mají formu knihovny paralelizačních rutin, probíhají především v režii výrobců paralelních počítačů, ty jsou pak upravovány pro danou platformu. Existují však i platformně nezávislé a vesměs také volně dostupné kvalitní balíky, které může kdokoliv použít na svém počítači, aniž by se nutně muselo jednat o víceprocesorový stroj.

V minulosti proběhl v této oblasti výzkum, zabývající se implemenetací MPI přímo do hardware systému, kdy by MPI operace byly prakticky zabudovány do mikroobvodů čipů paměti RAM v každém uzlu. Takový typ implementace by byl nezávislý na programovacím jazyku, operačním systému nebo dokonce na procesoru v systému, ale nemohl by být snadno aktualizovatelný, či odstranitelný.

Z paralelních knihoven využívajících MPI, které jsou k dispozici, lze uvést například MPIX Library, MPI Cubix, ScaLAPACK, Cluster Graphic Library a další. Ačkoliv jazyk Java nemá žádný oficiální vztah se standardem MPI, bylo zde již několik pokusů provázat Javu s MPI s různými stupni úspěšnosti. Jeden z prvních pokusů byl mpiJava od Bryana Carpentera. V podstatě jde o zabalení C MPI knihovny do Java Native Interface, což vyústilo v hybridní implementaci s omezenou přenosností. Nicméně tento původní projekt, známý také jako mpiJava API byl brzy následován dalšími Java MPI projekty. Jako méně užívaná alternativa je zde MPJ API, navržená tak, aby více odpovídala objektově orientovanému programovaní a kódovým konvencím společnosti Sun Microsystems. Další knihovny, mimo API, mohou být buď odvozené od místní MPI knihovny nebo mohou implementovat vlastní funkce založené na modelu předáváni zpráv, z nichž některé, jako P2P-MPI Java poskytují peer to peer funkční komunikaci i mezi různými platformami (například smíšené Linux a Windows klastry). Některé z nejzajímavějších částí implementace založené na MPI jsou pro Javu ztraceny kvůli omezením a charakteru samotného jazyka, jako je de-facto absence pointerů a lineární adresace místa v paměti, což činí přenos mnohorozměrných polí nebo velmi

složitých objektů neefektivním.

Standard MPI se neustále vyvíjí a pracovní skupina MPI Fóra je pořád aktivní. Zatím poslední verze standardu MPI, MPI-2.1 byla vydána 9. února 2007. Veškeré dokumenty projektu jsou dostupné na stránkách MPI fóra (viz [8]). Vzhledem k dlouhému používání a vývoji, který vychází a dědí rysy z předchozích verzí, je existence a používání MPI garantováno na mnoho let dopředu. Volné tělo standardu, a komerční produkty, které potřebují MPI, zajišťují i v budoucnu aktivní vývoj tohoto projektu.

3 Implementace paralelní úlohy

Při seznamování s jazykem Java mi pomohla literatura [1]. Po porozumění základním mechanismům v Javě jsem mohl přistoupit ke zpracovávání příkladů, které by mi pomohly technologie serializace, více-vláknového zpracování procesů a RMI pochopit tak, abych je mohl případně v závěru projektu ověřit v praxi.

3.1 Serialiace objektů

Na prvním místě bych chtěl uvést příklad na serializaci objektů. Použil jsem již existující program (viz [3]), do kterého bylo třeba připsat kód, který instanci třídy `Integral` v polovině výpočtu serializuje a vzápětí znovu obnoví. Jako ukázku (tabulka 1) uvádím část kódu třídy `NITest`, která, jakožto spustitelná, serializaci provádí. Jen dodám, že třída `Integral` musí samozřejmě implementovat rozhraní `Serializable`:

```
ObjectOutputStream  vystup = new ObjectOutputStream
(new FileOutputStream("integral.out"));
        vystup.writeObject(i);
        vystup.close();
        ObjectInputStream vstup = new ObjectInputStream
        (new FileInputStream("integral.out"));
        Object integral = vstup.readObject();}}}
```

Tabulka 1: Uložení instance objektu do souboru a její obnovení

3.2 Multithreading

Technologii vícevláknového zpracování procesů jsem prozkoumal pomocí programu složeného ze tří tříd, z nichž je ta první uvedena v příkladu na konci kapitoly (tabulka 2). Jak je vidět, třída `ThreadTest` vytvoří tři instance třídy `ThreadCisla`, která je rozšířena třídou `Thread`, a jednu instanci třídy `ThreadAbeceda`, která implementuje rozhraní `Runnable`, kde každá instance představuje jedno vlákno připravené k běhu. V případě rozšíření třídou `Thread` stačí pouze zavolat příslušné metody `start()` pro spuštění vlákna a `join()`, když chceme zjistit, zda vlákno už provedlo všechny své instrukce a skončilo. Pokud ale, jako v případě třídy `ThreadAbeceda`, implementujeme rozhraní `Runnable`, musíme tyto metody sami napsat, například, aby bylo možné kontrolovat doběh vlákna:

```
public void zkontroluj() throws InterruptedException {
t.join();
}
```

Program tedy funguje tak, že spustí všechna vlákna, kdy každé vlákno vypíše na obrazovku svůj název, a postupně číslo od jedné do deseti. Pokud je spuštěn na více-procesorovém stroji, je skutečně vidět paralelní běh vláken.

```
public class ThreadTest {
   public static void main(String[] args) {
      System.out.println( "ThreadTest - Start" );
      ThreadCisla tc0 = new ThreadCisla( 10 );
      ThreadCisla tc1 = new ThreadCisla( 10 );
      ThreadCisla tc2 = new ThreadCisla( 10 );
      ThreadAbeceda ta0 = new ThreadAbeceda( 10 );
      tc0.start();
      tc2.setPriority(10)  ;
      tc1.start();
      tc2.start();
      ta0.start();
      ...
```

Tabulka 2: Spustitelná třída

3.3 RMI

K seznámení s technologií RMI jsem použil zdroj z internetu (viz reference [7]). Díky tomuto poměrně jednoduchému návodu jsem si uvědomil, co vše je při používání technologie RMI potřeba. Ukázkový program mi pomohl se všemi úpravami které jsou potřeba při spuštění RMI systému na jednom počítači. Po spuštění serveru registru `rmiregistry` lze spustit server a následně již je možné používat metody, které jsou na serveru k dispozici, v klientském programu. Vše je sice spouštěno na jednom počítači, ale celý systém by se choval stejně, i kdyby byly jeho části spojené přes síť. V tomto případě klient pouze zjistí datum, které je na serveru, vypíše ho na obrazovku a informuje uživatele o úspěšnosti připojení. Ukázky kódu uvedu až v podkapitole 3.4, jelikož server je stále tentýž a mění se pouze klient, deklarace metod v rozhraní, a implementace metod v implementační části.

3.4 Program

Cílem práce bylo ověřit možné využití paralelizačních schopností jazyka Java. K tomu byl vybrán prostředek lehce popsatelný a mnohokrát zdokumento-

vaný, tedy násobení matic. Nejprve bylo potřeba navrhnout a implementovat třídu, která by samotné operace s maticemi zajišťovala. Samotný algoritmus násobení je poměrně jednoduchý (viz tabulka 7), ale mimo to je nutné zajistit mnoho dalších operací, jako vygenerování matice, její rozdělení na vhodné části, systém rozesílání a po obdržení výsledku i technika složení výsledné matice z dílčích částí. Jako další prvek byla sestavena třída, která zajišťuje celý systém vláken - jejich vytvoření, spuštění a kontrolu doběhu. Nakonec vznikl systém RMI, který celý program zastřešil a umožnil testování modelu jako paralelního distribuovaného systému na klastru Hydra.kai.tul.cz, ale i na jakémkoliv víceprocesorovém počítači.

3.4.1 Třída Matrix

Nejprve je nutné mít vůbec nějaké matice, které by bylo možné mezi sebou násobit. O to se stará metoda `generatorM` (tabulka 3).

```
public static Matrix generatorM( int k, int l ){
       Matrix c = new Matrix( k, l );
       for ( int i = 0; i < k; ++i ){
           for ( int j = 0; j < l; ++j ){
               c.setA( i, j, ( Math.random() * 10 - 5 ) );
           }
       }
       return c;
   }
```

Tabulka 3: Generátor matic

Metoda je statická, proto k jejímu volání není třeba instance objektu Matrix. Dvě vstupující proměnné `k` a `l` určují rozměry budoucí matice, kde `k` je počet řádků a `l` je počet sloupců. Program nejprve zavolá konstruktor `Matrix` (tabulka 4), jež založí objekt typu `Matrix` a alokuje v něm pole typu double požadovaných rozměrů. Nyní metoda vygeneruje $k * l$ náhodných čísel typu double v rozmezí $0 - 5$ a na každou pozici právě vytvořené matice jedno uloží. Tak máme vytvořen objekt typu Matrix, který v sobě obsahuje náhodně vygenerovanou matici, se kterou je již možné dál pracovat.

```
Matrix( int i , int j ) {
       a = new double[ i ][ j ];
       ai = a.length;
       aj = a[ 0 ].length;
   }
```

Tabulka 4: Konstruktor třídy Matrix

Vygenerovanou matici je třeba rozložit na dílčí celky, aby násobení mohlo proběhnout ve více částech najednou. Po několika pokusech jsem se rozhodl provést rozdělení první matice po řádcích (metoda `splitByNRows`) a rozdělení druhé matice po sloupcích (metoda `splitByNCols`). Matice je tak rovnoměrně rozdělena na určitý počet bloků, kde každý obsahuje rovnoměrný počet řádků při nezměněném počtu sloupců. Obě metody fungují stejně, proto se dále budu zabývat vysvětlením funkce pouze metody `splitByNRows`. Jediný vstupující parametr určuje, po kolika se budou řádky stávající matice rozdělovat na segmenty. Vystupujícím produktem této metody je pole objektů typu Matrix, z nichž každý v sobě obsahuje jednu část rozdělené matice. Algoritmus nejprve vydělením známého počtu řádků v matici požadovaným počtem řádků na segment zjistí, jak velké pole objektů `Matrix` bude třeba. Ošetřen je i případ, kdy by bylo požadováno třeba matici o sedmi řádcích rozdělit na bloky po třech řádcích. V metodě (tabulka 5) se první část algoritmu věnuje bezproblémovému celočíselnému rozdělení (například matici o šesti řádcích rozdělit do dvou bloků po třech řádcích) a druhá část řeší problémovější rozdělení - spočítá si, kolik celých segmentů lze z matice dostat a zbylé řádky vloží do posledního, menšího segmentu.

```
public Matrix[] splitByNRows( int cislo ){
      int vejdeSe = ai/cislo;
      if ( (vejdeSe * cislo ) == ai ){
          Matrix[] c = new Matrix[ vejdeSe ];
          for ( int i = 0; i <  vejdeSe ; i++ )
              c[ i ] = getNRows( i * cislo, cislo );
          return c;
      } else{
          int zb = (ai - ( vejdeSe * cislo ) );
          Matrix[] c = new Matrix[ vejdeSe + 1 ];
          for ( int i = 0; i <  vejdeSe; ++i )
              c[ i ] = getNRows( i * cislo, cislo );
          c[ vejdeSe ] = getNRows( vejdeSe  * cislo, zb );
          return c;
      }
  }
```

Tabulka 5: Metoda splitByNRows

Metoda ještě používá pomocnou metodu `getNRows` (tabulka 6), která pomocí vstupních parametrů `idx` a `nR` určí jaký a jak velký blok řádků z matice vybrat, a tento blok vrátí volající metodě.

Metoda `multiply` vynásobí mezi sebou dvě matice obsažené v objektech `Matrix`, které do metody vstupují jako parametry. Algoritmus násobení dvou matic je skutečně poměrně jednoduchý, jak je vidět v tabulce 7.

Metoda není statická, výsledek násobení průběžně zapisuje (pokud je to třeba i přepisuje) do objektu, pro který je volána.

```
Matrix getNRows(  int idx,  int nR  ){
    Matrix  mr = new Matrix( nR, aj );
  for( int i = idx; i < idx + ( nR ); i++){
      for ( int j = 0 ;  j < aj ; j++  )
  mr.setA(( i -  idx ) , j  , a[ i ][  j ] );
   }
    return mr;
}
```

Tabulka 6: metoda getNRows

```
void multiply( Matrix b, Matrix c ) {
       double pom;
       if( b.cols() == c.rows() ){
           for ( int l = 0 ; l < b.rows() ; ++l ){
               for ( int j = 0 ; j < c.cols() ; ++j ){
                   pom = 0;
                   for( int i = 0 ; i < b.cols() ; ++i ){
                       pom += b.getA( l, i ) * c.getA( i , j );
                       a[ l ][ j ] = pom;
                   }
               }
           }
       } else System.out.println( "Matice nelze nasobit" );
   }
```

Tabulka 7: Metoda multiply

Po vynásobení (metoda `multiply` je volána pro pole objektů typu `Matrix`) máme dílčí výsledky uloženy v předem připraveném poli jako segmenty výsledné matice. Konečné složení segmentů do výsledné matice zajišťuje metoda `putTogehther`. Vsupním parametrem je dvourozměrné pole objektů typu `Matrix`, ze kterého si metoda vezme každou položku a umístí ji na správné místo do výsledné matice, kterou také vrací. Algoritmus ošetřuje mnoho různých možností, a proto je v tabulce 8 uvedena jen jeho základní část, která zařizuje umístění pouze neokrajových segmentů.

Třída ještě obsahuje několik metod používaných k ladění. Než bylo možné přejít k samotnému testování délky výpočtu v různých situacích, bylo nejprve nutné ověřit, zda počítání opravdu funguje, proto vznikla metoda `equals` (tabulka 9), jejímž vstupním parametrem je objekt typu `Matrix`. Metoda není statická, proto byla volána pro instanci objektu `Matrix`, ve kterém se nacházel ověřený výsledek násobení matic, a která vracela výsledek `true` (pokud jsou matice v objektech naprosto shodné), nebo `false`. Vstupním parametrem byl složený výsledek z násobení proběhlého pomocí vláken. Tak bylo možné ověřit funkčnost operace násobení (rozložení na segmenty, výpočet v jednotlivých vláknech) a také správná funkčnost metody `putTogether`.

```
public Matrix putTogether( Matrix[][] a ) {
        int r = a.length;
        int s = a[ 0 ].length;
        int rows = a[ 0 ][ 0 ].rows();
        int cols = a[ 0 ][ 0 ].cols();
        int rowsL = a[ r - 1 ][  s - 1 ].rows();
        int colsL = a[ r - 1 ][ s - 1 ].cols();
        Matrix c = new Matrix( ( ( r -1 )*rows ) +rowsL, ( ( s -1 )*cols ) +colsL );
        for ( int i = 0; i < ( r -1 ); ++i ){
            for ( int j = 0; j < ( s - 1 ); ++j ){
                for ( int k = 0; k < rows; ++k ){
                    for ( int l = 0; l < cols; ++l ){
                      c.setA((i*rows)+ k,  (j*cols)+ l, a[ i ][ j ].getA(k,l));
                    }
                }
            }
        }
...
```

Tabulka 8: Ukázka metody putTogether

```
public boolean  equals(  Matrix  b ){
boolean check = true;
double eps = Math.pow( 10, -5 );
try{ for ( int i =0; i  < ai; ++i ){
for ( int j =0; j < aj; ++j ){ if ( check == false ) break;
                    if ( Math.abs( getA( i, j ) - b.getA( i, j )) > eps){
                        check = false;
                        break;
                    }
                }
            }
        }catch ( ArrayIndexOutOfBoundsException e ){
            check = false;
        }
        return check;
    }
```

Tabulka 9: Metoda equals

3.4.2 Třída MatrixMultiplicator

Třída implementuje rozhraní `Runnable`, které umožňuje práci s vlákny, a také rozhraní `Serializable`, aby byla vlákna následně použitelná v RMI. Díky možnosti skládání tříd v Javě, může třída `MatrixMultiplikator` používat metody třídy `Matrix`, aniž by ji rozšiřovala. Konstruktor třídy (tabulka 10) přijme ve vstupním parametru dva objekty typu `Matrix`, v nichž jsou předem připravené segmenty matic připravené k násobení, a vytvoří nový objekt typu `Matrix` pro uložení výsledku, ve kterém alokuje pole patřičných rozměrů (počet řádků první vstupující matice a počet sloupců druhé). Následně vytvoří instanci třídy `Thread`, pomocí které se později budou provádět operace s vlákny.

```
public MatrixMultiplicator( Matrix a,  Matrix b ) {
       mm = new Matrix( a.rows(), b.cols() );
       this.a = a;
       this.b = b;
       t = new Thread( this );
}
```

Tabulka 10: Konstruktor třídy MatrixMultiplicator

Metoda `start` spustí vlákno, metoda `join` kontroluje, zda již vlákno skončilo svůj běh, a konečně metoda `run` (tabulka 11) říká vláknu, co má za svého běhu provádět. V tomto případě vlákno vytvoří instanci třídy `TimeCounter`, čímž zároveň spustí měření času, dále algoritmus provede vynásobení matic, a výsledek uloží do objektu `Matrix` připraveného k tomuto účelu již po zavolání konstruktoru třídy.

```
public void  run() {  double pom;
       TimeCounter time = new TimeCounter(); if( a.cols() == b.rows() ){
           for ( int l = 0 ; l < a.rows() ; ++l ){
               for ( int j = 0 ; j < b.cols() ; ++j ){
                   pom = 0.0;
                   for( int i = 0 ; i < a.cols() ; ++i ){
                       pom += a.getA( l, i ) * b.getA( i , j );
                       mm.setA( l, j, pom);
                   }
               }
           }
       }
       threadTime = time.threadTime();
}
```

Tabulka 11: Metoda run

Poslední metodou ve třídě je `GetProduct`, která vrací v konstruktoru vytvořený objekt typu `Matrix` nyní již obsahující výsledek násobení.

3.4.3 RMI System

RMI systém se skládá ze čtyř základních částí: klient, na straně klientského počítače, rozhraní, implementace rozhraní a server na straně hostitelského počítače. Samozřejmě, jak bylo uvedeno výše, klientský a hostitelský počítač mohou být totožné, záleží na tom, jak se nastaví adresa připojení v klientovi.

Klient má za úkol pomocí metod třídy `Matrix` vygenerovat matici, rozdělit ji na segmenty, rozeslat k výpočtu, převzít výsledek a složit ho do výsledné matice. Klient je ten, kdo voláním správných metod říká celému programu, co má jaká jeho část dělat, určuje také hodnoty proměnných (velikost matic, počet vláken a podobně). V našem případě má z důvodů testování dvě verze. Jedna verze (tabulka 12, `RMIMatrixClient2`) pošle na server vlákno, počká na jeho doběh, převezme výsledek, a až poté pošle další. Je zřejmé, že v tomto případě se nejedná o paralelní počítání, je to zde pouze z důvodu porovnání mezi sekvenčním a paralelním výpočtem.

```
      ....
try {
          for ( int i = 0; i < threads; ++i ){
                //serverName = "compute-" + 0 + "-" + i;
                myServerObject[ i ] = (RMIMatrixInterface) Naming.lookup
                ( "rmi://" + serverName + "/myRMIImplInstance" );
          }
          k = 0;
          for ( int i = 0; i < arrayH; ++i ){
              for ( int j = 0; j < arrayW; ++j ) {
                  idx[ k ] = myServerObject[ k ].multiply( ra[ i ], sb[ j ] );
                  myServerObject[  k ].setA( idx[ k ] );
                  System.out.println( myServerObject[ k ].getA() );
          System.out.println();
                  System.out.println( myServerObject[ k ].getA() );
                  myServerObject[ k ].join( idx[ k ] );
                  threadTimes[ k ] = myServerObject[ k ].getThreadTime( idx[ k ] );
                  mm[ i ][ j ] = myServerObject[ k ].getProduct( idx[ k ] );
                  ++k;
              }
          }
      } catch( Exception e ) {
          System.out.println( "Exception occured: " + e );
          e.printStackTrace();
          System.exit( 0 );
      }
      ....
```

Tabulka 12: Sekvenční verze klientského programu

Nyní popíšu operace, které provádí kód v chráněném bloku `try`:

1. V prvním cyklu `for` navazuje klient spojení se serverem. Dokud testování probíhalo na klastru Hydra, měnila se proměnná `ServerName` (jejíž

implicitní hodnota nastavená na začátku programu je `locahost` - čili místní počítač) na hodnoty `compute-0-0` až `compute-0-n`, kde $n + 1$ je počet vláken potřebných k výpočtu. Tyto proměnné jsou názvy uzlů na klastru Hydra. K vytvoření instancí objektu `MyServerObject` dojde tak, že se klient "podívá" (lookup) na určenou adresu serveru, a pokud se mu dostane odpovědi, vytvoří s pomocí serveru instanci implementační třídy.

2. Druhý cyklus `for` má hned na začátku vnořen ještě jeden cyklus `for`. První cyklus cyklus určuje souřadnici v poli, vzniklého z první matice (rozdělení první a druhé matice po řádcích i po sloupcích proběhlo na začátku klienta) a druhý souřadnici v poli z druhé matice. Hned v následujícím kroku je volána metoda `multiplyT`, o které ještě bude řeč, a o které prozatím stačí vědět, že zajišťuje vytvoření a spuštění vlákna.

3. V tom samém cyklu se (zcela sekvenčně) počká na doběh vlákna (metoda `join`) a poté se hned převezme výsledek (metoda `getProduct`). Také se uloží čas běhu vlákna.

Klient, který provádí násobení paralelně (`RMIMatrixClient`), se od výše popsaného liší tím, že činnosti popsané v bodě 2. a 3. provádí ve dvou oddělených cyklech. Tedy nejdříve provede vytvoření a spuštění všech vláken, a až v dalším cyklu, kdy již všechna vlákna paralelně běží, se stará o to, zda již svou činnost dokončily a případně přebírá výsledky. Výsledky použití obou klientů budou blíže rozebrány v podkapitole 3.4.4

V rozhraní (`RMIMatrixInterface`) je deklarace metod, které jsou přístupné klientovi. Každá metoda musí mít za hlavičkou deklaraci `throws java.rmi.RemoteException`.

Server (`RMIMatrixServer`) má za úkol nastavení `SecurityManager` a také vytváří instanci třídy implementace, v níž jsou implementovány metody, deklarované v rozhraní, díky čemuž pak může klient tyto metody používat.

Implementace (`RMIMatrixImpl`), jak je zmíněno výše, obsahuje implementaci metod deklarovaných v rozhraní. Jsou to takové metody, které hostitel umožňuje používat klientovi. Na začátku je ve třídě deklarováno pole typu `MatrixMultiplicator`, pole má tolik prvků, na kolik vláken bude výpočet rozdělen. Konstruktor této třídy (jejíž instanci vytváří RMIMatrixServer) zajišťuje vazbu mezi klientským programem a programem serveru. Z metod uvedu jen jednu: `multiplyT` najde v připraveném poli typu `MatrixMultiplicator` první volnou pozici, a pro ni zavolá konstruktor této třídy, přičemž vstupujícími parametry jsou dvě matice určené ke znásobení. Vznikne tak vlákno, které je hned v dalším kroku spuštěno.

Dále je zde několik různých metod, pomocí kterých může klient pracovat s vlákny (získání výsledku, měření času…).

3.4.4 Testování

Vzhledem k tomu, že v době, kdy jsem měl se svou prací přistoupit k testování a měření časů, byl školní klastr Hydra dočasně mimo provoz, musel jsem tuto konečnou fázi provést ne svém osobním počítači. Jedná se o dvoujádrový procesor Pentium D945, s taktem jádra 3,4 GHz. Vzhledem k tomu že se jedná pouze o dvoujádrový procesor, nelze efekt dalšího zrychlení vidět již od dvou a více vláken. Bohužel program byl navržen pro výpočetní klastr o šestnácti uzlech, proto je schopen výpočet rozdělit pouze na 1, 4, 9 a 16 vláken.

Test byl prováděn se dvěma čtvercovými maticemi o rozměrech 1200 x 1200, jejichž výpočet jsem vždy rozdělil na 1,4 a 9 vláken. Nejprve byl pro test použit klient schopný paralelního spouštění vláken (`RMIMatrxKlient`):

```
1 vlákno:
Time of thread nr.0 is 0:01:4.609
Total time : 0:01:6.78
4 vlákna:
Time of thread nr.0 is 0:00:31.112
Time of thread nr.1 is 0:00:31.88
Time of thread nr.2 is 0:00:31.326
Time of thread nr.3 is 0:00:31.405
Total time : 0:00:32.969
9 vláken:
Time of thread nr.0 is 0:00:29.503
Time of thread nr.1 is 0:00:28.560
Time of thread nr.2 is 0:00:29.563
Time of thread nr.3 is 0:00:27.351
Time of thread nr.4 is 0:00:30.298
Time of thread nr.5 is 0:00:27.296
Time of thread nr.6 is 0:00:29.16
Time of thread nr.7 is 0:00:30.166
Time of thread nr.8 is 0:00:30.89
Total time : 0:00:32.360
```

Tabulka 13: Výsledky paralelního klienta

V tabulce 14 jsou vypsány časy vláken klienta, který vlákna odesílá procesoru sekvenčně.

V prvním případě odešle klient procesoru všechna vlákna najednou, a ty se poté střídají v běhu tak, jak jim operační systém přiděluje čas. V druhém případě klient odešle vždy pouze jedno vlákno a čeká na jeho doběh. I když je tedy samotné vlákno rychlejší (nemusí se dělit s ostatními o procesorový čas), celková doba výpočtu je delší. Nicméně, aby byl efekt paralelizace více patrný, bylo by zapotřebí více procesorů.

```
1 vlákno:
Time of thread nr.0 is 0:01:4.140
Total time : 0:01:5.515
4 vlákna:
Time of thread nr.0 is 0:00:14.985
Time of thread nr.1 is 0:00:14.922
Time of thread nr.2 is 0:00:15.140
Time of thread nr.3 is 0:00:15.78
Total time : 0:01:2.172
9 vláken:
Time of thread nr.0 is 0:00:06.250
Time of thread nr.1 is 0:00:06.250
Time of thread nr.2 is 0:00:06.234
Time of thread nr.3 is 0:00:06.250
Time of thread nr.4 is 0:00:06.250
Time of thread nr.5 is 0:00:06.250
Time of thread nr.6 is 0:00:06.234
Time of thread nr.7 is 0:00:06.266
Time of thread nr.8 is 0:00:06.250
Total time : 0:00:59.156
```

Tabulka 14: Výsledky sekvenčního klienta

Téměř dvojnásobný čas délky trvání výpočtu při zaslání pouze jednoho vlákna procesoru je dán tím, že se operace účastní pouze výpočetní výkon jednoho jádra, tedy poloviční síla. Jestliže je výpočet rozdělen na dvě a více vláken, operační systém se vždy snaží dělit práci rovnoměrně mezi obě jádra procesoru.

Závěr

Tato práce se zabývala problematikou paralelismu a využitím paralelních technologií, především z pohledu programovacího jazyka Java.

Práce byla rozdělena do tří tematicky oddělených kapitol.

V první kapitole je přiblížena problematika paralelismu, rozebrány základní principy. Prostor je věnován popisu paralelních architektur a teoretických zásad.

Druhá kapitola je zaměřena na popis technologií jazyku Java a standardu MPI.

Třetí kapitola obsahuje postup při tvorbě systému pro distribuované počítání s použitím technologií vícevláknového zpracování procesů, serializace a RMI. Na konci kapitoly jsou shrnuty výsledky praktické zkoušky systému na testovací úloze.

Práce otestovala paralelní technologie, které poskytuje programovací jazyk Java. Bylo dokázáno, že vytvořený systém lze úspěšně použít pro distribuování výpočtu na výpočetní klastr.

Reference

[1] B.Eckel: *Thinking in Java* 2^{nd} *Edition*, Grada Publishing, 2000

[2] Kenneth Baclawski: *Java RMI Tutorial*,
http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

[3] Dalibor Frydrych: *Cvičení k předmětu SRM*,
http://flow.kmo.tul.cz/ dalibor/srm/

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *jGuru: Remote Method Invocation*,
http://java.sun.com/developer/onlineTraining/rmi/RMI.html

[5] Andrew Downs: *Java Serialization*,
http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/

[6] McGraw-Hill/Osborne: *Multithreading in Java*,
http://www.devarticles.com/c/a/Java/Multithreading-in-Java/

[7] Tom Valesky: *How to create an RMI system*,
http://patriot.net//~tvalesky/easyrmi.html

[8] *MPI Forum*, http://www.mpi-forum.org/