TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1234T567 – Informační technologie

Java a paralelní programování na clusteru

Java and parallel programming on a cluster

Diplomová práce

Autor: **Bc. Miroslav Špetlák**
Vedoucí práce: Ing. Igor Kopetschke

V Liberci 21.5.2010

Prohlášení

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

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce.

Datum: 21.5.2010

Podpis:

Poděkování

Na tomto místě bych chtěl poděkovat a vyjádřit uznání všem, kteří mi pomáhali při psaní této diplomové práce. Především pak vedoucímu Ing. Igoru Kopetschke za zájem, věcné připomínky a čas, který mi věnoval při konzultacích.

Také bych chtěl poděkovat svým rodičům a slečně Haně Jiránkové za poskytnuté zázemí, za trpělivost a především za motivaci k dokončení diplomové práce.

Anotace

Cílem teoretické části diplomové práce je komplexní rozbor problematiky paralelního programování na clusterech se zaměřením na knihovny MPI, MPICH a PVM. Každá knihovna je podrobně popsána z hlediska historického vývoje, vlastností, jejích výhod a nevýhod. Dále je součástí práce popis programovacího jazyka Java jako vhodného nástroje pro paralelní programování. Jsou zmíněny různé implementace MPI technologie pro jazyk Java, včetně použitého MPJ Express. Vysvětlena byla problematika clusterů a také Amdahlův zákon, který definuje vztahy pro výpočet časové úspory v paralelním programování. Praktická část práce obsahuje návrh a implementaci vzorových úloh v jazyce Java pro použití na univerzitním clusteru Hydra. Tyto úlohy tvoří aplikace pro hledání hesla „hrubou silou“ a program pro paralelní násobení matic. Byla vytvořena sada srovnávacích testů pro jednotlivé úlohy za použití knihoven MPI, srovnávající běh aplikací na různém počtu procesorů. Analýzou výsledků srovnávacích testů byla ověřena platnost Amdahlova zákona.

Klíčová slova: paralelní programování, cluster, MPI, PVM, Amdahlův zákon.

Abstract

The theoretical part of this thesis is a comprehensive analysis of problems associated with parallel programming in clusters, with a focus on MPI, MPICH and PVM libraries. Each library is described in detail in terms of historical development, properties, and its advantages and disadvantages for parallel programming. There are different implementations of MPI technologies for Java described within, including the MPJ Express used. Issues with clusters are explained, as well as Amdahl’s law, which defines relationships for calculation of time savings in parallel programming. The practical part of this thesis includes the design and implementation of sample tasks in Java, to be used on a university cluster Hydra. These tasks form applications for searching passwords „by brute force“ and a program for parallel matrix multiplication. Set of comparison tests for individual tasks using MPI libraries have been created. These compare applications running on a different number of processors. Analysis of the comparative test results has validated the Amdahl’s law.

Keywords: parallel programming, cluster, MPI, PVM, Amdahl’s law.

Obsah

Úvod

Při paralelním zpracování úloh dochází k provádění dvou a více úloh současně. Téměř všechny operační systémy dovolují paralelní zpracování úloh pomocí jedné ze dvou následujících technik: paralelního zpracování na bázi procesů a paralelního zpracování na bázi vláken.

Paralelní zpracování na bázi procesů se stará o současné provádění více programů. Programátoři se na program odvolávají jako na proces. Proto lze říci, že paralelní zpracování na bázi procesů je paralelní zpracování na bázi programů.

Paralelní zpracování na bázi vláken znamená, že program provádí dvě úlohy současně. Například textový procesor může provádět kontrolu správnosti slov v dokumentu, zatímco my tento dokument píšeme. To je příklad vícevláknového paralelního zpracování.

Rozdíl mezi procesovým a vícevláknovým paralelním zpracováním si lze představit tak, že procesové zpracování pracuje s více programy a vícevláknové s několika částmi jednoho programu. Cílem paralelního zpracování je využít čas nečinnosti procesoru a zrychlit zpracování úloh.

Výše popsané techniky však nevytváří skutečný paralelismus, nýbrž takzvaný pseudoparalelismus, kdy se procesy vzájemně dělí o časová kvanta jednoho CPU. Pro potřeby této diplomové práce jsou však neméně důležité tzv. paralelní systémy (konkrétně clustery). Cílem této diplomové práce je podrobně zrešeršovat tyto paralelní systémy, jejich klasifikaci, využití, výhody a nevýhody a dále navrhnout sadu aplikací pro školní cluster HYDRA. Pomocí uvedené sady aplikací bude vyhodnocen přínos paralelizace úloh s využitím Amdahlova zákona.

1 Paralelní programování

1.1 Co jsou to paralelní algoritmy

Paralelními algoritmy rozumíme ty, u kterých samostatné procesy provádějí výpočet a zároveň interagují výměnou informací [3]. Paralelní jsou proto, že běží zároveň více procesů (výpočtů) a tím se odlišují od klasických sekvenčních výpočtů, u kterých probíhá v jednom momentu jenom jeden výpočet. Paralelní algoritmy mohou vznikat „paralelizací“ sekvenčních algoritmů, přičemž se hlavní problém rozdělí na podproblémy, které se pak v případě, že jsou na sobě nezávislé (na vyřešení jednoho problému nepotřebujeme řešení druhého problému) můžou řešit zároveň. Paralelizací se může výpočet výrazně zrychlit.

1. SIMD – (Single Instruction Multiple Data) – provádějí se stejné operace, ale na různých datech. Příkladem je například prohledávání grafu do hloubky, kde se používá stejný algoritmus, ale na různých částech dat
2. MISD – (Multiple Instruction Single Data) – provádění různých operací na stejných datech. V tomto případě se jedná například o různé statistiky, kde se vstupní data nemění výpočtem, takže mohou být zpracovávána více operacemi zároveň
3. MIMD – (Multiple Instruction Multiple Data) – různé operace na různých datech. Jedná se například o multitasking, kde více programů pracuje nezávisle na sobě i když u jednoprocesorových systémů se najednou provádí v jednom momentu pouze jedna operace.

MPI a PVM se používají hlavně pro SIMD a MIMD výpočty, jelikož MISD není závislý na vzájemné komunikaci procesů.

1.2 Komunikace mezi procesy

Při komunikaci se může jednat o jednostranný nebo kooperativní přenos dat. O jednostranný přenos dat jde v případě zápisu a čtení ze vzdálené paměti, kde k datům mohou procesy přistupovat nezávisle na sobě. O kooperativní přenos dat se jedná, pokud se na přenosu musí podílet obě strany – odesílatel i příjemce. Výhodou je, že u příjemce nedojde ke změně dat v jeho paměti bez jeho účasti.

Dalším důležitým faktorem pro návrh aplikací je znalost architektury, pro kterou jsou tyto aplikace určeny.

Podle architektury rozlišujeme následující paměťové modely:

1) Distribuovaná paměť
   - Používá se například u Paragon, IBM SPx, sítí pracovních stanic
   - Každý procesor má svou vlastní paměť
2) Sdílená paměť
   - Používá se například u SGI Power Challenge, Cray T3D
   - Procesory používají jednu a tu samou sdílenou paměť

Rozdíl mezi jednotlivými modely spočívá v přístupu k paměti a z toho plynoucích způsobech komunikace.

2 MPI – Message Passing Interface

Message Passing Interface (MPI) je specifikace standardu pro výměnu zpráv, pro použití u paralelních počítačů, clusterů a heterogenních sítí. Byla navržena pro podporu paralelních výpočtů vyžadujících kooperaci jednotlivých procesů. Tento standard má mnoho implementací jako jsou SUN MPI, LAM, MPICH, MPICH 2 apod.

2.1 Historie MPI

Začátek vývoje standardu MPI se datuje na březen 1992 na Message Passing Standardization Workshop ve Williamsburgu, MPI Fórum bylo zorganizováno na konferenci Supercomputing v listopadu 1992. Vývojáři se následujících 18 měsíců setkávali každých šest týdnů na dva dny a dojednávali jednotlivé části standardu. Na konferenci Supercomputing v roce 1993 byla vydána dočasná verze a finální verze 1.0 MPI standardu byla vydána v květnu roku 1994. Po malých úpravách byla na jaře 1995 vydána verze 1.1, která se používá i v současnosti pod označením MPI-1. Kompletním přepisem pak vznikl standard MPI-2, který byl stanoven za nástupce standardu MPI-1. Jednotlivé implementace MPI se liší také kombinací standardu MPI-1 a MPI-2.
Standard MPI byl specifikován MPI Fórem, které seskupovalo:

1) Širokou veřejnost
2) Vývojáře knihoven: PVM, p4, Zipcode, TCGMSG, Chameleón, Express, Linda
3) Společnosti: ARCO, Convex, Cray Res, IBM, Intel, KAI, Meiko, NAG, nCUBE, ParaSoft, Shell, TMC
4) Laboratoře: ANL, GMD, LANL, LLNL, NOAA, NSF, ORNL, PNL, Sandia, SDSC, SRC
5) Americké univerzity

Široká spolupráce při specifikaci MPI vedla k návrhu komplexního standardu, který flexibilně reagoval na potřeby zúčastněných subjektů, což vedlo k vysoké popularizaci MPI.

2.2 Vlastnosti MPI

Mezi základní charakteristiky MPI patří:

1) Komunikátory kombinující kontext a skupiny pro bezpečnost zpráv – veškerá komunikace probíhá v rámci daného komunikátoru, není umožněna komunikace mezi procesy, které používají rozdílné komunikátory.
2) Dvoubodová komunikace odstraňuje problémy se směrováním zpráv z pohledu programátora

   Charakteristiky komunikace:

   a) Strukturované buffery
   b) Odvozené datové typy
   c) Heterogenita komunikujících stran – umožňuje propojení počítačů s různými architekturami bez nutnosti řešit nekompatibilitu architektur z pohledu programátora
   d) Režimy komunikace
      i. Normální – blokující a neblokující
      ii. Synchronní – posílání a příjem musí probíhat zároveň
      iii. Ready – pro přístup k rychlým protokolům
      iv. Bufferovaná – asynchronní s využitím bufferů pro dočasné uložení zpráv
3) Kolektivní funkce umožňující provádění hromadných aktivit zahrnují:
   a) Vestavěné a uživatelské funkce pro sběr dat – data jsou sesbírána buď jediným procesem nebo všemi procesy zároveň, přičemž programátor se nemusí starat o adresaci zpráv
   b) Rutiny pro přesun dat
   c) Podskupiny definované přímo nebo dle topologie
4) Topologie procesů orientovaná na aplikace – vestavěná podpora pro gridy a grafy
5) Profilování – podpora přerušení volání MPI
6) Měření času – podpora pro sledování výkonu systému
7) Automatická konverze dat – vynucené typování dat všech funkcí zajišťujících komunikaci. Nespornou výhodou je odstranění problémů s datovou nekompatibilitou mezi některými architekturami výpočetních systémů.
8) Poskytuje thread-safe API – podpora bezpečnosti pro vícevláknové systémy

MPI zatím neumí:

1) Process management – procesy neumí vyvolávat, rušit ani spravovat jiné procesy, což vede k obtížnějšímu návrhu robustních aplikací schopných reagovat na výpadky v rámci systému
2) Vzdálené paměťové přenosy
3) Aktivní zprávy
4) Vlákna – obsahuje pouze podporu pro thread-safety
5) Virtuální sdílená paměť – více procesů požadujících stejná data zbytečně zatěžují systém požadavky komunikace

2.3 Implementace standardu MPI

Ze standardu MPI vychází několik implementací, které se liší rozsahem implementace (škálou funkcí), pracovním prostředím a syntaxí.

Mezi nejvýznamnější implementace patří:

1) MPICH – prostředí heterogenních sítí a multiprocesorových systémů se sdílenou pamětí
2) Prostředí pracovních stanic – limitováno funkcionalitou socketů v Unixu
   a) LAM
      - Vyvinuto v Ohio Supercomputer Center
      - Běží na heterogenních sítích Sun, DEC, SGI, IBM, HP pracovních stanic
   b) CHIMP-MPI
      - Vyvinuto v Edinburgh Parallel Computing Center
      - Běží na systémech jako LAM a navíc na Fujitsu AP-1000
   c) Unify
      - Vyvinuto v Mississippi State University
      - Vrstvy MPI nad upravenou verzí PVM umožňující kombinaci MPI a PVM volání

Některé implementace MPI jsou vyvíjeny jako open-source projekty, tudíž jejich zdrojové soubory jsou veřejně dostupné, avšak existuje řada implementací, které jsou součástí komerčních aplikací a nejsou tudíž volně dostupné pro veřejnost.

2.4 Základní funkce MPI

Základních funkcí knihovny je šest, přičemž již takový počet umožňuje napsat jednodušší aplikaci. První čtyři z těchto funkcí je třeba použít v každém MPI programu. Všechny funkce vrací celočíselný kód úspěšnosti provedené operace, přičemž symbolická hodnota MPI_SUCCESS znamená úspěch.

Přehled základních funkcí v syntaxi programovacího jazyka C (za použití [7]):

- `int MPI_Init (int* argc, char*** argv)`
  Inicializace MPI výpočtu, `argc` a `argv` jsou argumenty hlavního programu.
- `int MPI_Comm_size (MPI_Comm comm, int* adr_size)`
  Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem `adr_size`. `MPI_Comm` je typu „komunikátor“, pokud při volání dosadíme za parametr `comm` hodnotu `MPI_COMM_WORLD`, zjišťujeme počet všech vytvořených procesů aplikace N.
- `int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)`
  Zjištění čísla procesu v rámci „komunikačního světa“ `comm`. Čísla jsou v rozmezí 0 až M-1, kde M je rozměr komunikačního světa reprezentovaného komunikátorem `comm` (M = N, pokud dosadíme za `comm` hodnotu `MPI_COMM_WORLD`).
- `int MPI_Finalize (void)`
  Ukončení výpočtu v MPI, provádí každý proces.
- `int MPI_Send (void* adr_buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  Odeslání zprávy s typem `tag` v komunikačním světě `comm` procesu `dest`. Odesílá se zpráva z bufferu `buf` obsahující `count` položek typu `datatype`.
  Buffer je tedy pro zprávu, na rozdíl od PVM, kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například MPI_INT, MPI_DOUBLE, MPI_CHAR, nebo strukturované typy vytvořené z primitivních.
- `int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* adr_status)`

Blokující příjem zprávy. Parametry mají analogický význam jako u `MPI_Send`. Navíc je parametr `adr_status`, odkazující na místo, kam se má uložit status příjmu zprávy (výstupní parametr).

Status obsahuje položky:

`status.MPI_TAG` – jakého typu je příchozí zpráva

`status.MPI_SOURCE` – od koho je příchozí zpráva

Dosadí-li se za `source` hodnota `MPI_ANY_SOURCE` a za `tag` `MPI_ANY_TAG`, přijme se jakákoli zpráva a z uvedených položek výstupního parametru status se dá zjistit, jaká zpráva přišla.

3 MPICH

MPICH je volně dostupná kompletní implementace MPI specifikace, která byla navržena s ohledem na dosažení co nejvyšší přenositelnosti a efektivity. Označení MPICH bylo zvoleno jako zkratka MPI a písmene CH, označujícího slovo chameleón, které symbolizuje adaptabilitu k prostředí a tudíž přenositelnost. Jelikož chameleóni jsou rychlá zvířata, mají také charakterizovat druhotný cíl návrhu MPI – efektivitu a výkon. MPICH je zároveň výzkumným projektem a projektem pro vývoj software. Z pohledu výzkumného projektu bylo cílem prozkoumat metody využití prostoru mezi programátory paralelních počítačů a výkonem poskytovaným jejich hardwarem. Při návrhu byla zohledněna všechna omezení standardu MPI a vypuštěna všechna omezení architektury cílových počítačů při zachování vysokého výkonu (měřen jako propustnost dat a zpoždění operací pro výměnu zpráv). Z pohledu projektu pro vývoj software bylo cílem nabídnout vývojářům volně dostupnou, vysoce výkonnou, multiplatformní implementaci standardu MPI.

3.1 Historie MPICH

Vývoj MPICH probíhal již v době návrhu standardu MPI a tudíž poskytoval možnost v praxi odzkoušet jednotlivé části specifikace. Při vývoji MPICH byly použity a částečně přepsány součásti následujících systémů:

1) P4 – jedná se o knihovnu třetí generace pro podporu programování paralelních systémů obsahující komponenty pro výměnu zpráv a sdílení paměti přenositelné do mnoha prostředí paralelních výpočtů včetně heterogenních sítí. P4 také obsahuje podporu pro sítě postavené na protokolu TCP/IP a multiprocesorové systémy se sdílenou pamětí.
2) Chameleón je vysoce výkonný přenositelný balík pro výměnu zpráv na paralelních superpočítačích. Byl implementován jako tenká vrstva (většinou makra jazyka C) nad již existujícími systémy (NX, CMMD, MPL) pro výkon a nad volně dostupnými systémy jako jsou p4a PVM pro přenositelnost.
3) Zipcode je přenositelný systém pro psaní škálovatelných knihoven. Tento systém přinesl správu skupin procesů a virtuální topologie.

3.2 Architektura MPICH

Při návrhu MPICH byl kladen důraz na to, aby co nejvíce kódu bylo společného pro všechny implementace bez újmy na výkonu a také aby byl MPICH co nejrychleji přenositelný na nové architektury, na kterých by se předefinovaly společné funkce pro maximalizaci výkonu.

Abstract Device Interface (ADI) – všechny funkce MPI jsou implementovány jako makra a funkce, které jsou plně přenositelné. ADI obsahuje komplexní specifikaci, která pokrývá všechny operace, jež se mohou vyskytnout, což umožňuje mnoho implementací ADI v MPICH. To pak nabízí přenositelnost, snadnou implementaci a možnost volby mezi přenositelností a výkonem. Jedna z implementací ADI je channel interface, která obsahuje jen nejzákladnější funkce a umožňuje rychlý přenos MPICH do nového prostředí.

MPICH ADI nabízí následující skupiny příkazů:

1) Specifikování zprávy, která má být přijata nebo odeslána
2) Přesun dat mezi API a hardware
3) Správa seznamu zbývajících zpráv
4) Poskytování informací o běžícím prostředí

4 PVM – Parallel Virtual Machine

Parallel Virtual Machine (dále PVM) je softwarový systém, který umožňuje, aby se na heterogenní (z hlediska architektury, formátu dat, výpočetního výkonu, zatížení počítače a sítě, použitého operačního systému) kolekci počítačů nahlíželo z uživatelského hlediska jako na jeden paralelní počítač.

PVM je výsledkem výzkumného projektu výpočtů v heterogenních sítích, na kterém se podílejí Oak Ridge National Laboratory, University of Tenessee, Emory University a Carnegie Mellon University.

Jedná se o systém pro paralelní a distribuované výpočty, který podporuje přímý, ale funkčně kompletní message-passing model. PVM bylo navrženo pro propojení výpočetních zdrojů a poskytuje uživateli paralelní platformu pro spouštění aplikací nezávisle na počtu a umístění jednotlivých počítačů. Může se tak jednat o použití např. v rámci několika málo počítačů v laboratoři nebo o využití tisíců počítačů připojených do Internetu.

PVM nabízí možnost využití relativně levného výpočetního výkonu jednoprocesorových počítačů narozdíl od masivně paralelních procesorů (MPP) používajících stovky procesorů, jelikož jednoprocesorové systémy jsou univerzálnější (použitelné pro širší spektrum oblastí) a náklady na pořízení MPP sahají do stamilionových čísel. Pokud je PVM korektně nainstalován a konfigurován, poskytuje využití různorodých zdrojů (výpočetního výkonu, operační a diskové paměti) pro dosažení vysokého výkonu a funkcionality paralelních programů.

Existence jednoho paralelního počítače (dále VM) přináší pro programátora i uživatele mnoho výhod, mezi které patří:

1) Správa (spouštění, ukončení, kontrola běhu) všech procesů běžících v rámci jednoho VM z jednoho místa
2) Zjednodušená komunikace mezi procesy (automatické směrování zpráv, konverze dat)
3) Dynamické přidávání a odebírání počítačů ve VM
4) Využití „zbytkového“ výkonu počítačů
5) Podpora pro toleranci chyb – výpadky v síti, nedostupnost počítače, zatuhnutí počítače
6) Podpora pro load balancing

4.1 Historie a současnost PVM

Systém PVM prošel dlouhým vývojem od ryze laboratorního použití na univerzitách až po komerční nasazení.

Samotný začátek projektu se datuje na léto 1989 na půdě Oak Ridge National Laboratory, přičemž zpočátku se jednalo o vývoj prototypu budoucího systému. Stejně jako prototyp, tak i první verze PVM 1.0 vyvinutá Vaidy Sunderamem a Al Geistem nebyla veřejně dostupná, používala se výhradně v rámci laboratoře. První veřejnou byla až verze PVM 2.0 vyvinutá na University of Tenessee, která byla zveřejněna v březnu 1991. Sloužila k vědeckým aplikacím a prošla několika aktualizacemi, které byly reakcí na požadavky uživatelů, až do verze 2.4.

V únoru 1993 byl systém kompletně přepsán a tím vznikl PVM3. Zatím poslední je verze 3.4.5, přičemž existují i upravené verze jako je PVM s podporou pro Portable Batch System nebo PVM s rozdělenou serverovou a výpočetní částí. Nad samotným PVM byly postaveny nadstavby (CPPvm s podporou C++, hybrid PVM a MPI) a také optimalizované verze pro jednotlivé architektury a jádra systémů. Nechybí ani verze pro Windows.

4.2 Ideologie PVM

Použití PVM vychází z následujícího scénáře:

1) Návrh malých kooperujících aplikací
2) Pro vzájemnou komunikaci se použijí funkce z knihovny libpvm
3) Inicializace PVM spuštěním hlavního PVM démona (dále master pvmd)
4) Spuštění pvmd' odpovědného za spouštění slave pvmd
5) Připojení slave pvmd do PVM
6) Spuštění jednotlivých aplikací na slave pvmd
7) Správa PVM pomocí master pvmd

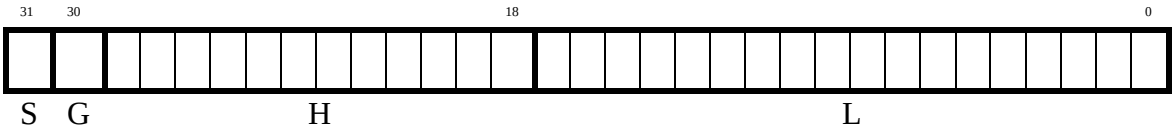
PVM je koncipováno jako centralizovaný systém, neboť spouštění slave pvmd a aplikací náleží master pvmd a tudíž je běh celého PVM závislý na běhu master pvmd. V případě výpadku slave pvmd nebo pvmd' lze spustit náhradní pvmd a reagovat tak na výpadek. Avšak v případě výpadku master pvmd dochází k ukončení celého PVM, proto je důležité, aby master pvmd běžel na spolehlivém počítači a zároveň se doporučuje používat tento počítač pouze k tomuto účelu. Celé PVM je přestavitelné za

běhu, to znamená, že systém může reagovat na vytížení jednotlivých počítačů a v závislosti na tom přesouvat úkoly mezi počítači a v případě potřeby spouštět nové pvmd.

4.3 Komponenty PVM

4.3.1 Task identifier

PVM používá task identifier (dále TID) pro adresaci jednotlivých pvmd, aplikací a skupin v rámci PVM.



Obr. 4.1: struktura TID

TID se skládá ze čtyř částí:

1) S-bit – používá se k adresaci pvmd
2) G-bit – používá se k multicastovým adresám (adresy skupin)
3) H – unikátní číslo pvmd
4) L – unikátní číslo procesu v rámci jednoho pvmd (samé nuly znamenají samotné pvmd)

Z toho vyplývá, že v PVM můžeme mít 2^{12} - 1 = 4095 počítačů a v rámci jednoho pvmd pak 2^{18} - 1 = 262 143 procesů.
V Unixu představuje hodnota L počítadlo. Mapování mezi L hodnotou a číslem procesu pak v systému zabezpečuje pvmd.

Použití TID	**S**	**G**	**H**	**L**
číslo procesu	0	0	1..4095	1..262 143
číslo pvmd	1	0	1..4095	0
lokální pvmd	1	0	0	0
pvmd' z master pvmd	1	0	0	0
multicast adresa	0	1	1..4095	1..262 143
chybový kód	1	1	Malé záporné číslo	

Tab. 4.1: Použití TID

4.3.2 Třídy architektur

PVM přiřazuje jméno architektury každému typu počítače, aby se rozlišilo mezi počítači, které používají různé spustitelné soubory a také aby se prováděla případná datová konverze.

4.3.3 Model zpráv

PVM používá zprávy, které obsahují data (typu char, int, atd.) a každá zpráva má takzvaný tag, tj. číslo, kterým se pak mohou odlišovat jednotlivé typy zpráv. Ke konverzi datových typů dochází automaticky v případě posílání zpráv mezi datově nekompatibilními architekturami. Odesílající strana nečeká na potvrzení přijetí, ale pokračuje, jakmile byla zpráva odeslána do sítě. Zprávy jsou bufferovány na straně příjemce do doby, než jsou přijaty. PVM spolehlivě doručuje zprávy, testuje existenci cílového počítače a garantuje pořadí zpráv. PVM poskytuje blokující (čeká, dokud nepřijme zprávu), neblokující (nečeká na příjem) a timeout příjem (čeká určitou dobu na příjem zprávy).

Ke komunikaci mezi pvmd se používá TCP protokolu, jelikož zaručuje doručování zpráv.

4.3.4 Asynchronní notifikace

PVM poskytuje podporu pro obnovu v případě výpadku pomocí takzvaných notifikačních zpráv. Jedná se o následující typy zpráv:

1) PvmTaskExit – aplikace skončila nebo selhala
2) PvmHostDelete – počítač byl odebrán z PVM nebo selhal
3) PvmHostAdd – do PVM byl přidán nový počítač

Tyto zprávy jsou zasílány v případě, že o ně požádá některý z počítačů v rámci PVM pomocí funkce pvm_notify(). V případě PvmTaskExit je zpráva zasílána pvmd, pod kterým sledovaná aplikace běží, PvmHostDelete je obsluhován pvmd, který zažádá o sledování stroje a PvmHostAdd je obsluhován master pvmd, jelikož ten se stará o správu PVM.

4.3.5 PVM démon

Na každém počítači v PVM běží jeden PVM démon (pvmd). V případě, že na jednom počítači pracuje více uživatelů, může běžet současně více pvmd v rámci

jednoho počítače, které pracují nezávisle na sobě. Pvmd slouží jako směrovač zpráv, zajišťuje místo spojení, autentizaci, správu procesů a detekci výpadku. Při stratu PVM je první pvmd prohlášen za master pvmd a všechny další jsou prohlášeny za slave pvmd. Master pvmd může plnit všechny funkce slave pvmd (spuštění aplikace, předávání zpráv, atd.) a navíc je odpovědný jako jediný v rámci PVM za přidávání a odebírání počítačů z PVM a v případě jeho ukončení končí práce celého PVM. V případě, že slave pvmd zažádá o rekonfiguraci PVM, je tento požadavek přeposílán master pvmd.

4.3.6 Knihovny pvm a gpvm

Součástí PVM instalace jsou i knihovny libpvm a lipgpvm. Libpvm umožňuje aplikaci komunikaci jak s pvmd, tak i s jinými aplikacemi, tj. funkce pro zabalování a rozbalování zpráv a funkce pro PVM systémové volání k obsluze pvmd. Knihovna libgpvm obsahuje funkce pro správu skupin procesů v rámci PVM.

4.3.7 Konzole pvm

Pro interaktivní správu PVM se používá konzole, spouštěná příkazem pvm z příkazové řádky. Při jejím startu je inicializováno PVM, tudíž v případě zadání tzv. hostfile můžeme inicializovat celé PVM včetně všech přidružených počítačů. Po spuštění máme pak možnost rekonfigurovat PVM, spouštět a ukončovat aplikace a ukončit celé PVM. Konzole je jenom nadstavbou PVM, tudíž všechny její funkce jsou dostupné i v uživatelských programech, až na inicializaci PVM.

4.3.8 XPVM

Součástí distribuce PVM je i nástroj XPVM, který kombinuje konzoli, sledování výkonu a jednoduchého debuggeru. Jedná se o grafickou aplikaci použitelnou pro grafickou reprezentaci práce PVM, sledování provozu v rámci PVM nebo pro inicializaci a správu PVM.

4.3.9 PVMRUN

Utilita pvmrun sice není součástí PVM, ale pro neinteraktivní spouštění aplikací je nejvhodnější. Jedná se o aplikaci, která se stará o inicializaci master pvmd a přidání dalších počítačů dle seznamu a spuštění požadovaného počtu kopií aplikace postavené nad PVM. Jeho použití je tedy možné jenom v případě, že spouštíme jednu aplikaci, ale

ve více kopiích. Rozhodnutí, na kterém počítači bude spuštěna zvolená aplikace, není v režii pvmrun, ale pvmlib, tudíž uživatel si může jen zvolit počet kopií aplikace ke spuštění a o zbytek se už postará pvmrun. V případě, že požadujeme spuštění více kopií než máme k dispozici počítačů, dojde ke spuštění více instancí v rámci jednoho pvmd. Pvmrun přesměruje výstupy všech aplikací na sebe a tudíž máme z počítače, na kterém se spouští pvmrun, možnost sledovat výstup ostatních zúčastněných počítačů. Z tohoto důvodu se nedoporučuje použití interaktivních aplikací. Pvmrun používá buď seznam počítačů z Portable Batch System (PBS), nebo z textového seznamu, přičemž konkrétní cesta k seznamu se uvádí před kompilací pvmrun.

4.3.10 Spouštění aplikací pod PVM

Jelikož součástí PVM je správa procesů, nabízí se nám několik možností jak spouštět aplikace:

1) Přímým zavoláním aplikace z příkazové řádky:
   Tímto způsobem dojde ke spuštění jedné instance, což je použitelné v případě, že chceme spouštět více různých aplikací. Příkladem mohou být master-slave programy, u kterých jsou master a slave část programu odděleně v různých spustitelných souborech.
2) Z již běžící aplikace pomocí funkce pvm_spawn:
   Touto cestou můžeme spouštět z jedné aplikace jinou aplikaci. Příkladem je konzole pvm, která tímto způsobem spouští zadané aplikace nebo program pvmDemoParent, který je master-slave programem, přičemž master je spouštěn z příkazové řádky a slave je vyvolán pomocí pvm-spawn v rámci běhu master aplikace.
3) Pomocí programu pvmrun – spuštění několika instancí jedné aplikace:
   Má podobnou syntaxi jako program mpirun, sloužící ke spuštění aplikací nad MPICH.

5 Programovací jazyk JAVA

5.1 Historie

Programovací jazyk Java, vyvinutý Markem Naughtonem, Mikem Sheridanem a Jamesem Goslingem ze Sun Microsystems, byl původně navržen pro řízení digitálních zařízení, jako například set-top boxů pro kabelovou televizi nebo zařízení PDA (Personal Digital Assistant) [5]. Java je programovací jazyk odvozený od jazyků C a C++, který je navržený jako jednoduchý, objektově orientovaný, nezávislý na architektuře, přenositelný, dynamický, distribuovaný, robustní a vícevláknový. Ačkoli prvotní iniciativa k vývoji Javy se objevila již v létě 1991, specifikace jazyka zůstává nejasná až do jara 1995, kdy společnost Netscape uzavírá smlouvu o začlenění technologie do nadcházejícího vydání jejího webového prohlížeče. Od té doby je jazyk přijímán širokou akademickou a vědeckou veřejností a také průmyslem, například společnostmi Intel, Microsoft, Novell, Oracle a dalšími firmami, které se rozhodly začlenit Javu do svých základních produktů. První Java Development Kit (soubor základních nástrojů pro vývoj aplikací) byl uveřejněn v lednu 1996 a netrvalo dlouho, než vývojáři vydali první aplikace založené na technologii jazyka Java.

5.2 Změny oproti starším programovacím jazykům

Jak již bylo zmíněno výše, Java je objektově orientovaný programovací jazyk založený na jazycích C a C++. Návrháři Javy měli za cíl vytvořit nástroj, který se vývojáři budou schopni rychle naučit, a který odstraní některé, pro programátory složité, konstrukce kódu C a C++. Příkladem takovýchto konstrukcí jsou: přetěžování operátorů, ukazatele, vícenásobná dědičnost, `goto` operátor. Přetěžování operátorů nově předefinovalo operátory +, -, *, / tak, že operátor volá metodu se souborem unárních nebo binárních parametrů. Ukazatele, které byly matoucí a byly příčinou mnoha programátorských chyb, byly nahrazeny referenčními typy. Referenční typy obsahují propojení s objekty Javy namísto odkazování na fyzické umístění v paměti. Rovněž byla odstraněna vícenásobná dědičnost, kdy třída dědila vlastnosti a metody od dvou nebo více rodičů. Příkaz „`goto`“ byl odstraněn a místo nich jsou používána klíčová slova „`break`“ a „`continue`“. Preprocesor jazyka C byl nahrazen balíčkovým systémem (packages). Struktury jazyka C byly také zcela odstraněny a nahrazeny třídami a silně typovými primitivami.

5.3 Zásadní výhody programovacího jazyka JAVA

Java je interpretovaným jazykem, přičemž překladač vytváří tzv. bytecode (též byte-kód) pro spuštění na virtuálním stroji (Java Virtual Machine - JVM), nevytváří se spustitelný soubor pro cílovou platformu. Zmiňovaný bytecode je nezávislý na architektuře, tudíž přenositelný (portabilní), a tak může být spuštěn na kterékoliv platformě, kde je dostupný virtuální stroj Javy (JVM). Java je silně typovým jazykem, kde reprezentace všech primitivních datových typů je přesně definovaná ve specifikaci jazyka (Java Language Specification) a překladač vyžaduje explicitní deklaraci všech metod. Java Virtual Machine je při běhu aplikace odpovědná za sledování rozsahu proměnných (array bound checking), zpracování výjimek (exception handling) a správu paměti (garbage collection).

Kromě výše uvedeného je Java také dynamickou, distribuovanou a vícevláknovou technologií. Java Virtual Machine je schopná dynamicky načítat třídy za běhu programu buď z lokálního úložiště, nebo ze vzdáleného místa přes síť Internet. Takzvané Java Native Interface je rozhraní umožňující propojit kód běžící na virtuálním stroji Javy s nativními programy a knihovnami napsanými v jiných jazycích – např. C, C++, apod., které jsou zkompilované pro určitý hardware, případně operační systém. Jedná se tedy o jakýsi převodní můstek, pomocí kterého se můžeme dostat za hranice virtuálního stroje. Introspekce (Introspection) umožňuje třídě Javy dynamicky za běhu vyhodnocovat informace o načtených třídách, například dostupné konstruktory, metody a vlastnosti. Remote Method Invocation určuje mechanismus, který poskytuje Java aplikacím možnost spouštět vzdálené objekty přes síť. Specifikace jazyka Java definuje podporu pro spouštění vláken, synchronizaci objektů, plánování vláken a skupiny vláken.

5.4 Bezpečnost

Java v sobě zahrnuje více bezpečnostních prvků oproti tradičním programovacím jazykům. Java Virtual Machine provádí verifikaci byte-kódu za účelem prevence proti nepovoleným bytovým kombinacím a také za běhu provádí „bounds check“ (kontrola překročení rozsahu polí) u všech prvků typu pole. Virtuální stroje integrované ve webových prohlížečích spouštějí applety v tzv. „sandbox“ modelu, kde je přístup k souborovému systému a síťové operace přísně omezen. Specifikace jazyka

neobsahuje podporu pro ukazatele, takže Java aplikace není schopna přistupovat do náhodné oblasti paměti mimo JVM. Referenční typy ukládají odkaz na objekty, které jsou dynamicky vytvářeny. Odkaz je použit virtuálním strojem pro získání vlastností a metod z hlavní paměti. Organizace a vývojáři mohou k aplikaci přiložit digitální podpis. Ten je použit k ověření integrity bytekódu, zabraňuje neautorizovaným modifikacím a stvrzuje hodnověrnost aplikace.

5.5 Výkon

Je známo, že výkon interpretovaných jazyků je horší než u kompilovaných. Java v tomto směru není výjimkou [5]. Aplikace napsaná v JDK verze 1.2 se vykonává zhruba desetkrát pomaleji než aplikace napsaná v jazyce C. Just-In-Time překladač, dostupný v některých implementacích Javy, překládá bytekód před vykonáním do jazyka cílového stroje. Just-In-Time překladač nesmírně zrychluje výkon většiny Java aplikací na úroveň srovnatelnou s aplikacemi psanými v jazyce C.

5.6 Serializace objektů

Programovací jazyk Java podporuje takzvaný Data Marshalling. Jedná se o shromáždění a transformaci dat do předem definovaného standardizovaného formátu před tím, než jsou odeslána přes síť. V Javě je tento proces nazýván serializací objektů. Spočívá v převodu objektu jazyka Java na pole bytů. Analogicky – deserializace objektu je převod pole bytů na objekt. Serializace objektů podporuje i celé seznamy (grafy) objektů, například může být serializován celý obsah spojového seznamu. Objekty tak mohou být uloženy do souboru a později znovu otevřeny pro další použití, případně mohou být odeslány přes síť s využitím protokolu TCP/IP, kde si objekt převezme jiná Java aplikace. Některé specifické části objektu mohou být označeny za dočasné (`transient`) – tyto části budou při serializaci přeskočeny, neboť při deserializaci by jejich kontext nemusel vyjadřovat přesný smysl. Knihovny pro serializaci objektů mohou být přizpůsobeny překrytím metod `readObject()` a `writeObject()`. Například uživatelská serializace by mohla dovolit programátorovi vytvořit ještě před serializací kompaktní pole bytů a po deserializaci spočítat správný počet prvků. Takzvaný Object Versioning předchází konfliktům způsobeným tím, že starší verzí objektu je serializována novější verze toho samého objektu.

5.7 Remote Method Invocation (RMI)

Remote Method Invocation je nástroj, který programátorovi umožňuje nabízet služby lokálního objektu vzdáleně přes počítačovou síť. Programátor definuje rozhraní objektu, toto rozhraní pak definuje signatury metod objektu, které jsou dostupné vzdáleně. Implementace třídy je potom založena na tomto rozhraní. Překladač RMI (RMIC – RMI Compiler) je použit k vygenerování spojovacích informací k dané třídě, generují se tzv. client stubs a server skeletons. Ve chvíli, kdy server chce zpřístupnit lokální objekt pro vzdálený přístup přes RMI, uloží rozhraní a dříve zmiňovaný server skeleton do tzv. registru. Předtím než je klient schopen zavolat metodu, musí přes registr zjistit lokaci objektu. Pokud byl objekt registrován, vrátí se klientovi příslušný client stub. Tento client stub poskytuje informace, jak komunikovat s daným objektem. Pokud klient přistupuje k objektu, server skeleton řídí tok dat mezi registrem a serverem. K objektům serveru může být také přistupováno přes speciální URL (Uniform Resource Locator) ve tvaru "`rmi://hostname[:port]/ObjectName`".

6 Implementace MPI pro programovací jazyk JAVA

6.1 Java jako přirozený nástroj pro paralelní programování

Potenciál Javy jako programovacího jazyka užívaného vývojáři pro psaní paralelních aplikací běžících na clusterech je obrovský. Java obsahuje mnoho nástrojů a bohatou sadu knihoven, které z ní dělají přirozený nástroj pro vývoj paralelních a distribuovaných aplikací.

Prvním důvodem, proč je Java tak vhodná pro psaní distribuovaných paralelních programů, je portabilita – přenositelnost. Zdrojový kód Javy je přeložen do byte-kódu nezávislého na architektuře, který může běžet na každém stroji, kde je k dispozici Java Virtual Machine. Navíc, Java Language Specification přesně definuje, jak jsou reprezentovány primitivní datové typy. Tato kompletní specifikace primitivních datových typů odstraňuje problémy spojené s konverzí datových typů na heterogenních virtuálních strojích.

Druhým důvodem, proč je Java přirozeným nástrojem pro paralelní programování, je jednoduchost programovacího jazyka. Java odstranila mnoho syntaktických konstrukcí, které činily kódy nepřehlednými a byly zdrojem častých programátorských chyb. Příkladem může být to, že Java již nepoužívá ukazatele. Programátoři pracující v jazyce C často strávili mnoho času laděním programu, kde ukazatel nedopatřením odkazoval na špatnou oblast paměti. Java nahradila ukazatele referenčními typy a silně typovým kontrolním systémem. Ten ověřuje, zda nebyl při běhu překročen rozsah polí a zda garbage collector uvolňuje paměť používanou objekty, pokud již aplikace nemá žádný odkaz na tyto objekty.

Třetím důležitým důvodem je, že Java zahrnuje funkcionalitu pro zjednodušení vývoje distribuovaných paralelních aplikací. Onou funkcionalitou je serializace objektů. Umožňuje odeslat uživatelsky definovanou datovou strukturu vzdálenému procesu pomocí volání jedné metody. Důležitou součástí je také Remote Method Invocation, pomocí této funkcionality může proces volat metodu vzdáleného stroje stejným způsobem, jako by tak činil lokálně. Serializace objektů a RMI jsou důležité komponenty pro MPI architektury vyvíjené pro Javu.

6.2 JavaMPI

Prvním pokusem o implementaci MPI-1 standardu pro Javu za použití JDK 1.0.2 bylo JavaMPI. Bylo vyvinuto na University of Westminster Savou Mintchevem a vydáno na sklonku roku 1997. JavaMPI je sada obalových funkcí, které umožňují přístup k existující nativní implementaci MPI jako například MPICH a LAM, za použití Native Method Interface obsažené v JDK 1.0.2. Native Method Interface poskytuje přístup k funkcím a knihovnám vytvořeným v jiných programovacích jazycích jako například v C nebo ve Fortranu. Obalové funkce byly vygenerovány pomocí JCI, nástroje pro automatické vytvoření obalové funkce z nativních metod Javy.

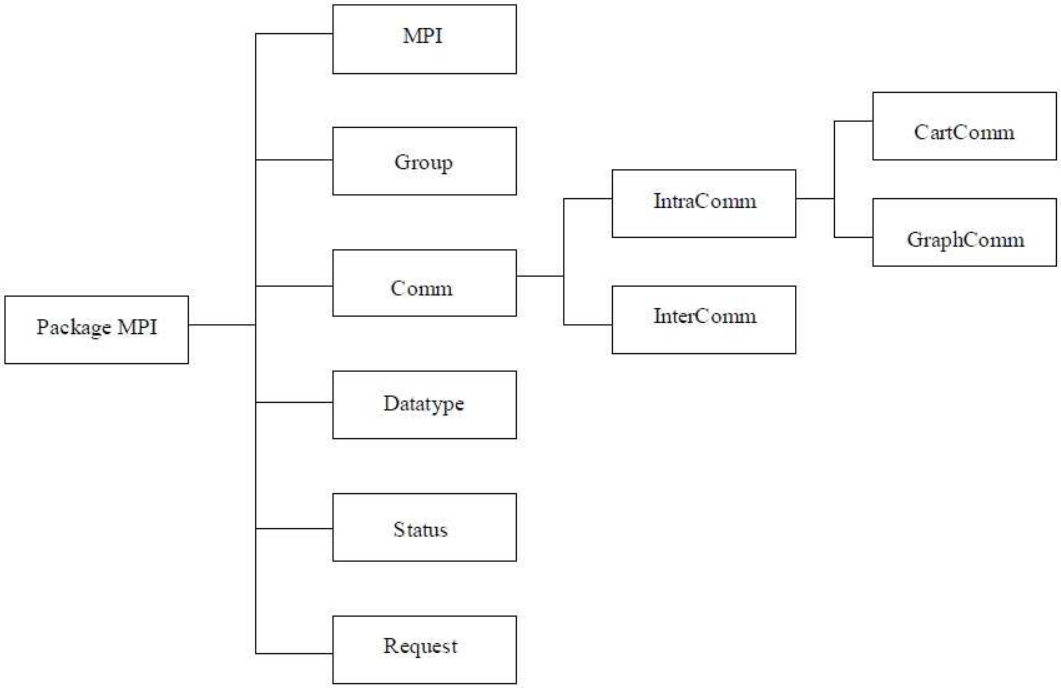
JavaMPI se skládá ze dvou tříd, MPIconst a MPI. Třída MPIconst obsahuje deklarace všech konstant MPI a také metodu `MPI_Init()`, která inicializuje prostředí MPI. Třída MPI obsahuje všechny ostatní funkce. Implementace MPI pro jazyk C obsahuje několik funkcí, které vyžadují argumenty předávané odkazem. Jelikož Java nepodporuje předávání odkazem, při volání funkce MPI s parametrem předávaným odkazem je vyžadováno použití objektů. To však komplikuje vývoj aplikace ve dvou směrech: u objektů musí být před použitím vytvořena nová instance ještě před zavoláním funkce MPI a také to, že hodnota proměnné je přístupná pouze pomocí vlastností objektu.

Jelikož v Javě nejsou podporovány aritmetické ukazatele, nelze odeslat prvek pole jako argument funkce tak, jak to lze v jazycích C nebo Fortran. Programátor může použít funkci z JCI: `JCI.section(arr, i)` – pro odeslání i-tého prvku z pole `arr`. Stejně tak je odlišné odesílání dílčího rozsahu z pole. Pokud chce programátor odeslat pouze část pole jinému procesu, aplikace musí nejprve vytvořit odvozený datový typ a ten teprve odeslat. Implementace JavaMPI výrazně změnila strukturu MPI aplikací. Výsledkem je, že přenos existující MPI aplikace na JavaMPI vyžaduje podstatné změny ve struktuře zdrojového kódu.

6.3 mpiJava

MpiJava poskytuje přístup k nativní implementaci MPI přes Java Native Interface. MpiJava bylo vyvinuto v Northeast Parallel Architectures Center na Syracuse University a bylo uveřejněno na sklonku roku 1998. Myšlenkou uplatněnou u mpiJava implementace bylo definovat svázání s MPI, které bude pro Javu přirozené. MpiJava model propojení s Javou je velmi blízký propojení s C++, jak bylo definováno ve

standardu MPI-2 a také podporuje MPI-1.1 standard. Hierarchie tříd je organizována stejně jako u implementace pro C++ definované v MPI-2 specifikaci a skládá se ze šesti hlavních tříd: MPI, Group, Comm, Datatype, Status a Request. Obrázek popisuje organizaci těchto šesti tříd.



Obr. 6.1: Organizace tříd v mpiJava (zdroj: [5])

Třída MPI je zde odpovědná za inicializaci a globální konstanty. Třída Comm definuje všechny metody komunikace jako odesílání a přijímání zpráv. První argument všech komunikačních metod vždy udává buffer, který má být odeslán nebo přijat. Implementace pro C a C++ požaduje počáteční fyzickou adresu s jedním nebo více prvky, které jsou MPI primitivního datového typu. Naproti tomu Java vyžaduje posílání objektů. Tento Java objekt bývá pole (o jednom nebo více prvcích) primitivního datového typu. Jelikož Java nepodporuje aritmetické ukazatele, všechny komunikační metody pro Java implementaci MPI zavádějí navíc parametr „offset“ – posunutí, které označuje počáteční prvek v poli.

Java nepodporuje volání parametrů odkazem, jediná realizovatelná možnost návratu hodnoty je pomocí metody „return“. Pokud MPI metoda změní prvky uvnitř pole, je vrácen počet změněných prvků. Jestliže tato metoda změní kompletně všechny prvky, počet změněných prvků metoda v tomto případě nevrací.

Výzkum v oblasti mpiJava se soustředí na využití serializace objektů. Jak již bylo uvedeno v kapitole 5.6., jde se o nástroj, který umožňuje vypsat stav objektu do souboru, případně ho odeslat přes síť jiné Java aplikaci. Stav takto serializovaného objektu potom může být obnoven takzvanou deserializací. V implementaci pro jazyk C je definován mechanismus, který popisuje rozvržení datové struktury jazyka C v paměti jako MPI odvozeného datového typu. Poté, co je datový typ definován, programátor může odeslat odvozený datový typ dalšímu procesu. Serializace objektů v Javě odstraňuje nutnost vytvoření odvozeného datového typu pro přenos vlastností objektu mezi MPI procesy. Programátor může jednoduše odeslat objekt komunikační rutině a nechat serializaci objektů, aby se postarala o vše potřebné.

6.4 MPJ Express

Pro účely diplomové práce byl použit systém MPJ Express, který je nainstalován na univerzitním clusteru Hydra.

Tento software je referenční implementací MPI definovanou pro programovací jazyk Java. Současná verze MPJE splňuje specifikaci mpiJava 1.2 API. V dohledné době se plánuje přidání podpory pro MPJ API. Rozdíl mezi těmito dvěma API implementacemi je v pojmenování tříd a metod. Poskytované funkcionality jsou pro uživatele stejné u obou API [9].

Existují dva způsoby spuštění MPJE aplikací. První a doporučený způsob je spuštění aplikace použitím „MPJ Express runtime infrastructure“, druhým způsobem je spouštění procesů manuálně.

MPJ Expres infrastruktura se skládá z démonů a modulu mpjrun. Myšlenka je taková, že uživatel nejprve spustí démony na několika výpočetních uzlech, které se budou podílet na běhu paralelní aplikace. V kontextu této diplomové práce to znamená, že démon je spuštěn na uzlech výpočetního clusteru Hydra. Jakmile jsou démony na jednotlivých uzlech spuštěny, uživatel může použít mpjrun modul (příkazem mpjrun.sh nebo mpjrun.bat) na hlavním uzlu clusteru. Modul se spojí se všemi démony, spustí MPJE program a výstupy přesměruje zpět na hlavní uzel. MPJ Express je schopen spouštěn jak JAR soubory, tak i soubory typu class.

Druhým způsobem spouštění, který byl výše zmíněn jako manuální, je spustit skript runmpj.sh, který používá SSH protokol k vykonávání programu. Tímto způsobem mohou být spouštěny JAR i class soubory, ovšem skript je možné použít pouze na

unixových operačních systémech. Pro operační systém Windows manuální spuštění aplikací znamená spustit každý MPJE proces pomocí Java příkazové řádky.

MPJ Express se v současné verzi nijak nezabývá bezpečností. MPJE démony mohou být bezpečnostním rizikem, jelikož se jedná o aplikaci naslouchající na určitém portu. Je proto doporučeno, aby démony běžely za vhodně konfigurovaným firewallem, což by zajistilo naslouchání pouze ověřeným důvěryhodným strojům. Mělo by být tedy zajištěno, aby démony běžely na uzlech clusteru, které nejsou přístupné z vnějšího světa. Nebo je možné tomuto předejít manuálním spuštěním MPJE procesů, k čemuž není zapotřebí aktivních démonů.

MPJ Express aplikace mohou být spouštěny ve dvou konfiguracích:

- Konfigurace Multicore – tato konfigurace je použita, pokud chceme spustit paralelní program na vícejádrovém počítači, nebo na stroji se sdílenou pamětí (laptopy a desktopy).
- Konfigurace Cluster - tato konfigurace je použita, pokud chceme spustit paralelní Java aplikaci na platformě s distribuovanou pamětí (clustery).

Spuštění MPJ Express aplikací v konfiguraci Multicore:

1) Stáhnout MPJ Express a rozbalit
2) Nastavit MPJ_HOME a PATH systémové proměnné
3) Napsat MPJ Express program a uložit (např. HelloWorld.java)
4) Přeložit: `javac -cp .;%MPJ_HOME%/lib/mpj.jar HelloWorld.java`
5) Spustit: `mpjrun.bat -np 4 HelloWorld.java`

Spuštění MPJ Express aplikací v konfiguraci Cluster:

1) Stáhnout MPJ Express a rozbalit
2) Nastavit MPJ_HOME a PATH systémové proměnné
3) Napsat MPJ Express program a uložit (např. HelloWorld.java)
4) Vytvořit a uložit soubor „`machines`“ obsahujícího jména (host names) nebo IP adresy všech uzlů clusteru, které chceme použít pro paralelní zpracování programu
5) Spustit démony. V případě univerzitního clusteru Hydra příkazem „`mpjboot machines`“

6) Přeložit: `javac -cp .;%MPJ_HOME%/lib/mpj.jar HelloWorld.java`
7) Spustit: `mpjrun.bat -np 4 -dev niodev HelloWorld`
8) Ukončit činnost démonů. V případě univerzitního clusteru Hydra příkazem „`mpjhalt machines`“

7 Clustery

Pokud v oboru informačních technologií hovoříme o clusteru, jedná se o propojení volně vázaných počítačů, které společně komunikují a vůči okolí se mohou tvářit jako jediný systém. Jednotlivé počítače jsou obvykle propojeny počítačovou sítí a pomocí clusterování lze dosáhnout vyšší výpočetní rychlosti nebo spolehlivosti s větší efektivitou, než které by bylo dosaženo pomocí jedné speciálně řešené výpočetní stanice. Clustery se využívají při paralelním zpracovávání úloh nebo jsou vhodné pro zajištění vysoké dostupnosti jednotlivých služeb (např. přístup do databáze, e-mailový server, webový server atp.). Pro vytvoření clusteru jsou dnes většinou používány víceprocesorové stanice propojené datovou sítí. Jednotlivé služby, které mají být zpracovány pomocí clusterování, musí být na provoz v clusteru přizpůsobeny.

Druhy clusterů

Existuje několik typů clusterů. Jednotlivé funkce a vlastnosti clusterů se prolínají, aby bylo dosaženo optimálních parametrů:

a) Cluster s vysokou dostupností (High-availability, HA) – pomocí několika počítačů je zajištěno nepřetržité vykonávání určité služby i v případě výpadku jednoho nebo více serverů z důvodů hardwarové závady či plánované údržby. Službu poskytuje jeden počítač, který je v případě výpadku automaticky zastoupen jiným počítačem.

b) Výpočetní cluster (High-performance computing, HPC) – cluster slouží ke zvýšení výpočetní rychlosti více počítačů, které spolupracují na společném výpočtu. Obvykle jsou použity počítače střední cenové hladiny propojené pomocí vysokorychlostní počítačové sítě. Tímto způsobem vznikne vysoce výkonný celek, který je mnohonásobně levnější, než jeden vysoce výkonný počítač.

c) Cluster s rozložením zátěže (Load balancing, LB) – snížení míry zátěže je dosaženo tím, že jednotlivá služba je poskytována několika počítači zároveň. Tyto počítače mají stejný obsah a služba tak je poskytována paralelně. Stejný obsah je zajištěn replikací obsahu mezi všechny propojené počítače nebo existencí specializovaného centrálního úložiště.

d) Úložný cluster (Storage cluster, SC) – tento typ clusterování zprostředkovává přístup k diskové kapacitě, která je rozložena mezi jednotlivé pracovní stanice.

Toho je dosahováno speciálními souborovými systémy, které jsou schopny zajistit rozložení zátěže, redundanci dat, pokrytí výpadků jednotlivých uzlů, distribuovaný mechanismus zamykání souborů a další doprovodné služby.

Pro účely této diplomové práce je důležitý typ výpočetního clusteru, který byl použit pro testování. Výpočetní clustery slouží k paralelním výpočtům složitých početních úloh, např. faktorizace na prvočísla, simulace vývoje počasí, analýza velkého množství statistických dat, atp. Úlohy určené pro urychlení za pomoci výpočetního clusteru musí být speciálně navrženy.

7.1 Univerzitní cluster Hydra

Na Fakultě mechatroniky, informatiky a mezioborových studií je umístěn výpočetní cluster Hydra. Jedná se o dvanáct dvouprocesorových (dvoujádrové procesory) stanic Dell PowerEdge 1950 a sedmnáct dvouprocesorových (jednojádrové procesory) stanic Sun Fire V20z. Jednotlivé stanice jsou mezi sebou propojeny síťovým rozhraním o rychlosti 1 Gbps. Na jednotlivých stanicích je nainstalován 64-bitový operační systém Linux Pentos ve verzi 5.4.
Technické parametry výpočetního clusteru Hydra:

- 12 uzlů Dell PowerEdge 1950 (též frontend)
    - 2x Intel Xeon 5140 2.33GHz/4MB 1333FSB (2 jádra)
    - 4GB RAM 667MHz (4x1HB)
    - 80GB SATA2, 7200 ot./min, hot plug
    - 2x NIC 1 Gbps, Broadcom NetXtreme II 5708 Gigabit Ethernet NIC
    - DVD ROM
    - Celkem: 24 CPU, 48 jader, 48GB RAM, 960GB HDD
- 17 uzlů Sun Fire V20z (1U)
    - 2x AMD Opteron 252, 2600 MHz (1 jádro)
    - 4 GB RAM
    - 73 GB HDD, 10025 ot./min, Fujitsu MAT3073NC
    - 1x Dual Ultra320 SCSI, LSI Logic 53c1030 PCI-X
    - 2x NIC 1 Gbps, Broadcom BCM5704
    - DVD-ROM, FDD
    - Celkem: 34 CPU (34 jader), 68 GB RAM, 1.2 TB HDD

- Rocks Clusters 5.2.2
- Linux CentOS 5.4
    - 64 bitový systém
    - binárně kompatibilní s Red Hat Enterprise Linux 5
- propojovací síť 1 Gbps
- přímá 1 Gbps konektivita do Liane, resp. Cesnetu

Jak se připojit

K Hydře (hydra.kai.tul.cz) je možné se přihlásit pomocí SSH. Z MS Windows se doporučuje použít klienta PuTTY (případně WinSCP pro nahrávání souborů na cluster). Pokud nechce uživatel při každém přihlášení zadávat svoje heslo, použije připojení pomocí veřejného klíče.

Uživatelské jméno se zadává ve standardním tvaru užívaném na Technické univerzitě v Liberci, to znamená „jmeno.prijmeni“ (například: miroslav.spetlak).

Informace o stavu clusteru je možné získat na adrese http://hydra.kai.tul.cz/ganglia/. Je ovšem nutno podotknout, že v průběhu zpracovávání diplomové práce byla funkčnost tohoto nástroje velmi omezená, na některých uzlech nebyl Ganglia démon spuštěn správně, tudíž nebylo možné se na výstupy spoléhat.

Při prvním přihlášení se automaticky vytváří dvojice klíčů, která umožňuje přístup z frontendu (řídící uzel clusteru) na nody (podřízené počítače pro výpočty). Veřejný klíč je automaticky uložen do `~/.ssh/authorized_keys`. Pokud je použita prázdná heslová fráze, bude se uživatel z frontendu Hydry moci přihlašovat na uzly bez zadání hesla stejně jako bez zadávání hesla na uzlech spouštět procesy pomocí příkazu cluster-fork.

Domácí adresáře uživatelů jsou umístěny na frontendu Hydry (tj. hydra.kai.tul.cz). Tento adresář se sdílí mezi všemi nody, takže pokud se na některý z nich uživatel přihlásí, bude mít všude k dispozici stejná data. Nody se jmenují compute-0-0 až compute-0-10 a compute-1-0 až compute-1-16.

8 Amdahlův zákon

Amdahlův zákon je pojmenován po Gene Amdahlovi, americkém počítačovém odborníkovi. Je používán pro určení maximálního možného zdokonalení celého systému, pokud jsou vylepšeny pouze některé části systému. Často se používá v paralelním programování pro předpověď teoretického maximálního zrychlení za použití paralelních systémů.

Zrychlení aplikace v paralelním programování je limitováno časem, potřebným k vykonání sekvenční části programu. Například, pokud program potřebuje dvacet hodin pro běh na jednojádrovém procesoru a z toho běží jednu hodinu kód, který nemůže být paralelizován (tudíž 19 hodin = 95 % kódu může být paralelizováno), potom bez ohledu na počet procesorů použitých pro paralelní část nemůže být celková doba vykonávání programu nižší než ona jedna hodina sekvenčního kódu. Proto může být aplikace zrychlena maximálně 20x.

Amdahlův zákon je model vyjadřující vztah mezi očekávaným zrychlením paralelizovaného programu a sekvenčním algoritmem za předpokladu, že úkol aplikace se paralelizací nezmění. Například pokud 12 % kódu aplikace může v paralelizovatelné podobě běžet „libovolně“ rychle (se zvyšujícím se počtem procesorů) a zbylých 88 % není paralelizovatelných, potom Amdahlův zákon říká, že maximální zrychlení paralelizované verze je 1 / (1 – 0,12) = 1,136x oproti neparalelizované verzi.

Amdahlův zákon definuje P jako část programu, která může být paralelizována a (1-P) jako část, která se musí vykonat sekvenčně. Potom maximální zrychlení za použití N procesorů je dáno vztahem:

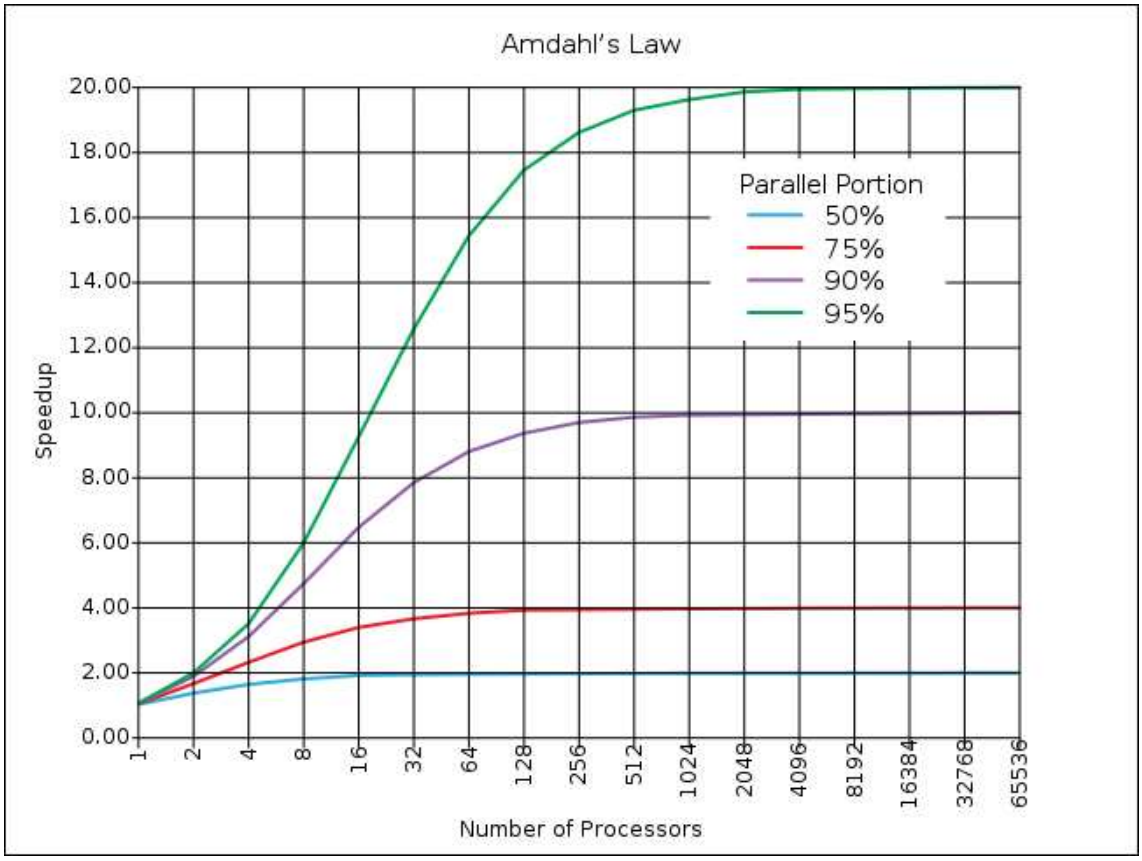
$$\frac{1}{(1-\mathrm{P})+\frac{\mathrm{P}}{\mathrm{N}}}$$

Pokud se N limitně blíží nekonečnu, potom se dá vztah vyjádřit jako:

$$\frac{1}{1-P}$$

Pro názornost: pokud `P = 90 %`, potom `(1-P) = 10 %`. Pokud dosadíme do výše uvedeného vztahu, zjistíme, že maximálně může být program zrychlen 10x, přičemž nezáleží na počtu použitých procesorů. Z toho vyplývá, že paralelní programování má smysl buď pro menší počet procesorů, nebo pro řešení problémů, kde je vysoká možnost paralelizace, tedy veličina `P`. Proto je v paralelním programování vždy velice důležité co nejvíce redukovat sekvenční část programu, čili veličinu `(1-P)`.

Ideální průběh zrychlení podle Amdahlova zákona je vidět na následujícím obrázku.



Obr. 8.1: Amdahlův zákon (zdroj: http://www.hardwarezone.com)

9 Navržené sady úloh

Pro testování výkonnosti výpočetního clusteru Hydra byly zvoleny dvě základní aplikace. Tyto aplikace byly dále parametrizovány tak, aby byla vytvořena sada srovnávacích testů, na nichž lze ověřit platnost Amdahlova zákona na paralelní clustery. První základní úlohou vhodnou pro paralelní programování je takzvaný Brute Force Attack. V podstatě se jedná o lámání hesel „hrubou silou“. Myšlenka spočívá v tom, že jsou postupně zkoušeny všechny kombinace možných znaků až do té doby, než je nalezen správný tvar hesla.

Druhou úlohou typickou pro podobné testy je násobení matic. Ať už se jedná o násobení matice vektorem, nebo násobení dvou matic, tyto výpočetní úlohy se velmi často používají při podobných testech výpočetních clusterů, ale i pro testování výkonnosti procesorů. Pro účely této práce byla zvolena varianta násobení matice maticí.

9.1 Brute Force Attack - SIMD

Řešení hrubou silou je způsob řešení problému či úlohy, při kterém se systematicky prochází celý prostor možných řešení problému. Jeho výhodou je nalezení opravdu nejlepšího řešení či případný důkaz o nemožnosti řešení problému. Jeho nevýhodou bývá velká složitost hledání, a tedy časová, případně paměťová náročnost algoritmu. Velmi často čas potřebný k nalezení řešení roste exponenciálně či dokonce s faktoriálem, takže i pro velmi malé prostory možných řešení je tato metoda v praxi nepoužitelná.

Někdy se jako řešení hrubou silou nesprávně označují také slovníkové útoky při prolamování hesel. Ty ale prohledávají již velmi zredukovaný prostor možných řešení a zároveň negarantují nalezení řešení.

V tomto případě jde o paralelní úlohu typu SIMD – Single Instruction Multiple Data. To znamená, že jsou prováděny stejné instrukce, ale na různých datech.

Při návrhu aplikace byl použit systém prohledávání všech možných řešení. To znamená, že bylo určeno heslo, které pak program měl naleznout. Zkoušeny byly všechny možné kombinace znaků z množiny povolených znaků. Testování bylo omezeno na maximálně pětiznaková hesla. Při vyšším počtu znaků hledaného hesla se časová náročnost testů již neúměrně zvyšovala. V tabulce 9.1. je pro názornost uveden počet možných variant při různé maximální délce hesla. Množina povolených znaků

obsahovala velké i malé znaky anglické abecedy a číslice, což činí dohromady 62 znaků.

Maximální délka hesla	Počet možných variant
1	62
2	3 906
3	242 234
4	15 018 570
5	931 151 402
6	57 731 386 986
7	3 579 345 993 194
8	221 919 451 578 090

Tab. 9.1: Počet možných hesel při různé maximální délce

Vzhledem k tomu, že hledání pětiznakového hesla trvá v průměru cca 10 minut (liší se v závislosti na parametrech úlohy), nalezení šestiznakového hesla, u něhož je 62x vyšší počet možných variant, by trvalo několik hodin. Z důvodu počtu prováděných měření byly tedy zvoleny varianty s hledáním hesla o čtyřech a pěti znacích.

9.1.1 Popis aplikace

Jakýmsi vstupem aplikace je programátorem zadaný řetězec, jehož přesnou podobu bude muset aplikace nalézt. Tento řetězec je konstantou aplikace a je vždy stejný po celou dobu testování. Po spuštění programu se vygeneruje několik souborů (jejich počet odpovídá počtu použitých uzlů), do nichž se postupně vypíší všechny možné varianty řetězců vzniklých z množiny povolených znaků. Varianty se vypisují dokud není dosaženo hodnoty pro maximální délku řetězce. Výpis do souborů probíhá tak, že se řetězce neukládají za sebou do jednoho souboru, nýbrž každý další následující řetězec je zapsán od jiného souboru. To v praxi znamená, že varianty jsou rozmístěny rovnoměrně a nikoliv způsobem takovým, že například hesla začínající jedním a tímž písmenem by byly umístěné všechny v jednom souboru. Tento výpis probíhá buď sekvenčně a nebo paralelně, v závislosti na prováděném typu úlohy.

Jakmile jsou všechny možné varianty hesla vypsané v souborech, program vypočítá MD5 hash programátorem zadaného hesla. Poté začne prohledávání všech souborů, program postupně načítá řetězce, vytváří jejich MD5 hashe a porovnává je s hashem původního hesla. Tato část je vždy vykonávána paralelně. Pokud je nalezena

shoda, program vypíše výsledek do konzole a ukončí se. Zároveň je poskytnuta informace o době potřebné k běhu aplikace.

Kompletní zdrojový kód je přiložen k diplomové práci na CD.

9.1.2 Parametrizace aplikace

Za účelem vytvoření sady testovacích úloh byla aplikace pro Brute Force Attack parametrizována dle různých hledisek. Byly vytvořeny dvě různé varianty paralelizace úlohy. Varianta s vyšší paralelizací provádí paralelně jak výpis všech možných řetězců do souboru, tak i následné hledání správného řetězce. Varianta s nižší paralelizací provádí paralelně pouze tu část, kde probíhá vyhledávání v souborech. Dále se složitost jednotlivých úloh dělí podle toho, jak dlouhé heslo se hledá. Jak již bylo uvedeno výše, zvoleny byly varianty pro čtyř-znakové a pěti-znakové heslo. Testy byly prováděny v obou režimech, které podporuje MPJ Express, to znamená v režimu Cluster a v režimu Multicore (jednotlivé režimy popsány v kapitole 6.4). Celkem tak bylo vytvořeno osm různých úloh, které pak byly postupně testovány na různém počtu stanic (1 – 27).

	4 znaky	**5 znaků**
Multicore	vyšší paralelizace nižší paralelizace	vyšší paralelizace nižší paralelizace
Cluster	vyšší paralelizace nižší paralelizace	vyšší paralelizace nižší paralelizace

Tab. 9.2: Parametrizace úloh pro Brute Force Attack

9.2 Násobení matic - MIMD

Jako druhá aplikace, vhodná pro testování clusteru, bylo zvoleno násobení velkých matic. Podobné úlohy se velmi často používají při demonstraci výhod paralelního programování a zároveň při testování výkonnosti paralelních počítačů a clusterů.

Aplikace byla navržena tak, aby fungovala jako paralelní úloha typu MIMD – Multiple Instruction Multiple Data. To znamená, že se provádějí různé instrukce nad různými daty.

9.2.1 Popis aplikace

Vstupními daty aplikace jsou dva soubory, kde jsou uloženy vstupní matice. Načítání ze souboru je nutné z toho důvodu, aby vstupní matice byly stejné po celou dobu testování. Výsledky testů tak nejsou ovlivněny různorodostí vstupních dat. Prvky matice jsou typu float a nabývají hodnot 0 až 9. Soubory byly vygenerovány náhodným algoritmem. Před přeložením a spuštěním aplikace programátor zadá rozměr matice N (buď 2000 nebo 3000) a v závislosti na této hodnotě jsou načteny vstupní matice. Načítání probíhá buď sekvenčně nebo paralelně, podle typu testu. Po kompletaci vstupních dat je matice A rozdělena na stejně velké úseky, které jsou rozeslány na jednotlivé uzly. Matice B je pomocí metody Broadcast rozeslána vcelku. Následně je spuštěn algoritmus samotného násobení matic, který vždy probíhá paralelně. Poté probíhá sběr výsledků z jednotlivých uzlů a kompletace matice C, která je uložena do výstupního souboru. Zároveň jsou použity funkce k měření času potřebného pro běh aplikace.

```
int chunkSize = (N) / size;
MPI.COMM_WORLD.Bcast(B,0,N*N,MPI.FLOAT, root);
MPI.COMM_WORLD.Scatter(A, 0, chunkSize*N, MPI.FLOAT, Asub, 0,
chunkSize*N, MPI.FLOAT, root);

// Násobení matic-----------------------------------
// Algoritmus násobení
//-------------------------------------------------

MPI.COMM_WORLD.Gather(Csub, 0, chunkSize*N, MPI.FLOAT, C, 0,
chunkSize*N, MPI.FLOAT, root);
```

Výňatek ze zdrojového kódu ukazuje rozeslání matice B pomocí metody `Bcast` na všechny uzly. Zároveň je matice A rozdělena na stejně velké úseky, které jsou také rozeslány na všechny používané uzly. Po ukončení výpočtu na zúčastněných stanicích jsou výsledky z každé z nich odeslány na uzel `root`, který je sesbírá a zkompletuje výslednou matici C.

Kompletní zdrojový kód aplikace je přiložen k diplomové práci jako příloha na CD.

9.2.2 Parametrizace

Aplikace pro násobení matic byla podobně jako předchozí úloha parametrizována z různých hledisek. Výpočetní náročnost se liší podle velikosti vstupních dat. Již bylo zmíněno, že jde o čtvercové matice, konkrétně byly zvoleny velikosti 2000x2000 a 3000x3000 prvků. Stupeň paralelizace je dán tím, zda načítání matic ze vstupních souborů je prováděno sekvenčně, nebo paralelně. U varianty s nižší paralelizací je prováděno sekvenčně, u varianty s vyšší paralelizací je pak načítání hodnot ze vstupních souborů prováděno paralelně. Stejně jako u Brute Force Attacku bylo i maticové násobení testováno v obou režimech MPJ Express – to znamená Multicore a Cluster.

	Matice 2000*2000 prvků	**Matice 3000*3000 prvků**
Multicore	vyšší paralelizace nižší paralelizace	vyšší paralelizace nižší paralelizace
Cluster	vyšší paralelizace nižší paralelizace	vyšší paralelizace nižší paralelizace

Tab. 9.3: Parametrizace úloh pro maticové násobení

10 Srovnávací testy

Motivací pro vznik této práce bylo ověření několika tezí spojených s paralelním programováním. Zejména bylo cílem ověřit platnost Amdahlova zákona, který mimo jiné říká, že maximální zrychlení paralelizované aplikace je limitováno tou částí programu, která není paralelizovatelná. Z toho vyplývá, že zrychlování paralelního programu není přímo úměrně závislé na zvyšování počtu použitých procesorů. Naopak, u některých paralelizovaných algoritmů se může stát, že režie pro obsluhu programu zvýší dobu potřebnou pro výpočet. Cílem tedy bylo vyvrátit domněnku, že při n-násobném zvýšení počtu hardwarových zdrojů lze očekávat n-násobné zrychlení. V praxi se toto děje pouze velmi vzácně, pokud však taková situace nastane, stává se tak díky neoptimálnímu navržení sekvenčního algoritmu.

10.1 Metodika měření

Při testování paralelních programů vyvstává otázka, jak měřit výkon algoritmu. Stanovení jednotného způsobu měření bylo velmi důležité pro interpretaci výsledků jednotlivých testů. Bylo třeba určit způsob měření sekvenčních algoritmů a jeho dílčích částí. Dále byla definována metodika pro měření paralelních programů. Důležitou součástí měření výkonnosti bylo určení míry paralelizace jednotlivých aplikací.

Měření sekvenčních algoritmů

Měří se doba, která uplyne od spuštění algoritmu až do jeho úplného ukončení. Měření probíhá pomocí nástrojů implementovaných do programovacího jazyka Java. Měří se s přesností na setinu sekundy.
Tuto dobu označme `Ts`.

Měření paralelních algoritmů

U paralelních algoritmů se měří doba, která uplyne od spuštění výpočtu až do doby, kdy skončí poslední ze všech paralelních výpočtů. Tento čas zahrnuje i distribuci vstupních a výstupních dat. MPJ Express obsahuje nástroje pro měření časových úseků v paralelních programech (metoda `Wtime`).
Dobu paralelního programu označme `Tp`.

Základní míra účinnosti paralelizace

Jedná se o poměr časů potřebných k vyřešení úlohy na jedné procesní jednotce a *p* procesních jednotkách. Co se týká sekvenčního algoritmu, tak vybíráme vždy čas toho nejvhodnějšího algoritmu. Tím se rozumí nejrychleji vykonaný sekvenční algoritmus. V praxi se často nesprávně používá čas potřebný k vykonání paralelního algoritmu spuštěného na jedné procesní jednotce [1].

Míru účinnosti paralelizace (zrychlení) označme S:

$$S = \frac{Ts}{Tp}$$

Určení míry paralelizace aplikace

Dostupná literatura týkající se problematiky paralelního programování přesně nedefinuje způsob určení míry paralelizace algoritmu. Pro účely této diplomové práce byla míra paralelizace určována poměrem časů jednotlivých částí programu určených k paralelizaci vůči času potřebnému k vykonání celého programu. To znamená, že při běhu sekvenčního algoritmu byly zvlášť měřeny úseky, které byly uvažovány jako paralelizovatelné. Poměr doby, potřebné k vykonání tohoto úseku a celkové doby, potřebné k vykonání sekvenčního programu, byl potom určen jako procentuální vyjádření paralelizovatelné části programu.

Pro názornost:

Celkový čas vykonávání sekvenčního programu T_s	100 sekund
Čas potřebný k vykonání úseku, který budeme paralelizovat T_1	75 sekund
Míra paralelizace aplikace P	75%

Tab. 10.1: Určení míry paralelizace

Jak již bylo zmíněno v předchozích kapitolách, takto paralelizovaná aplikace nebude nikdy rychlejší, než je čas potřebný k vykonání 25 % neparalelizovatelného kódu.

Samotné měření probíhalo na univerzitním clusteru Hydra. Každá varianta úlohy byla spuštěna třikrát, přičemž pokaždé byla zaznamenána doba běhu programu. Z těchto tří hodnot byla spočítána průměrná hodnota. Došlo tak k eliminování případných odchylek, například vlivem momentálně vytížených uzlů. Vytíženost clusteru je možné

sledovat přes webové rozhraní specializovaného softwaru Ganglia, ovšem v době zpracování diplomové práce byl tento velmi nespolehlivý. Ke zjištění vytíženosti clusteru byl tedy spíše používán jednoduchý skript (k práci je přiložen na CD jako součást příloh), který vypsal všechny aplikace běžící na jednotlivých uzlech. Bylo tak možné zjistit aktuální vytíženost clusteru. Samotná měření probíhala vždy v době minimálního vytížení.

Jak je uvedeno v kapitole 9, byly testovány dvě základní úlohy, každá v osmi různých parametrizacích. Měření probíhalo na 1 – 27 uzlech (jádrech), každé jednotlivé měření proběhlo třikrát, aby hodnota mohla být zprůměrována. Celkem tedy proběhlo cca 1300 měření.

10.2 Výsledky testů

10.2.1 Brute Force Attack

Aplikace pro hledání konkrétního textového řetězce „hrubou silou“ byla testována ve dvou základních variantách, které se odlišovaly délkou hledaného hesla. Vzhledem k časové náročnosti, která je vysoká i při paralelním zpracování, byly testovány řetězce o čtyř a pěti znacích. Takto dlouhé řetězce je možné nalézt při paralelním zpracování v čase řádově odpovídajícím sekundám až minutám. U hesel čítajících více znaků už časová náročnost stoupá řádově na hodiny, což by při počtu měření nebylo pro účely diplomové práce efektivní.

Zároveň byl vyvinut sekvenční algoritmus pro Brute Force Attack. Vůči němu pak bylo vztaženo vypočítané zrychlení a také podle něj byla určena míra paralelizace aplikace. Všechna měření byla provedena třikrát, poté byly použity průměrné hodnoty. Časy naměřené u sekvenčního algoritmu jsou uvedené v následující tabulce:

Typ úlohy	Měřený úsek	Čas [s]	%
4 znaky	celý sekvenční program	132,93	100
	vyhledávací algoritmus	121,95	91,74
	generování hesel do souboru	8,56	6,44
5 znaků	celý sekvenční program	2683,74	100
	vyhledávací algoritmus	2269,66	84,57
	generování hesel do souboru	365,36	13,61

Tab. 10.2: Podíly časů jednotlivých částí sekvenčního programu

Z tabulky je patrné, že procentuální podíl času potřebného pro vygenerování souboru roste se zvyšujícím se počtem znaků.

Předpokládané maximální zrychlení dle Amdahlova zákona

Aby mohlo být zjištěno předpokládané maximální zrychlení paralelizované aplikace, užijeme vztah vycházející z Amdahlova zákona:

$$\frac{1}{(1-P)}$$

kde P značí paralelní část programu a N je počet použitých procesních jednotek.

Paralelní verze programu se odlišují tím, že v první variantě je paralelizováno pouze vyhledávání řetězce v souborech vytvořených sekvenčně. Ve druhé variantě je navíc paralelizováno i vytvoření souborů, do nichž jsou uloženy všechny řetězce přicházející v úvahu pro danou délku hesla a množinu povolených znaků.

Dle Amdahlova zákona bylo určeno maximální možné zrychlení:

Typ úlohy	**Co bylo paralelizováno**	**Maximální možné zrychlení**
4 znaky	pouze vyhledávací algoritmus	12,1
	vyhledávání i generování hesel	54,85
5 znaků	pouze vyhledávací algoritmus	6,48
	vyhledávání i generování hesel	54,95

Tab. 10.3: Maximální možné zrychlení po paralelizaci

Tyto hodnoty byly porovnány se skutečnými naměřenými hodnotami:

	jednoduché heslo: Xj5R				**složitější heslo: puPi6**			
	nižší paralelizace		**vyšší paralelizace**		**nižší paralelizace**		**vyšší paralelizace**	
	čas[s]	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**
Multicore								
1	128,05	1,04	27,60	4,82	2554,60	1,06	3305,07	0,82
2	128,05	1,04	27,60	4,82	2554,60	1,06	3305,07	0,82
3	69,21	1,92	14,29	9,30	1569,82	1,72	1783,25	1,52
4	50,24	2,65	10,15	13,10	1183,10	2,28	1300,38	2,08
5	44,89	2,96	8,41	15,81	1063,73	2,54	1040,76	2,60
6	40,73	3,26	8,44	15,75	1037,08	2,61	1013,42	2,67
7	40,22	3,31	8,52	15,60	1028,52	2,63	1008,24	2,68
8	39,77	3,34	8,22	16,17	1014,77	2,66	971,24	2,78
9	42,37	3,14	8,73	15,23	1065,81	2,53	981,37	2,75
10	40,52	3,28	8,96	14,84	1024,95	2,64	976,67	2,77
11	39,75	3,34	8,54	15,57	1028,69	2,63	956,41	2,82
12	40,97	3,24	9,17	14,50	1020,62	2,65	1021,78	2,64

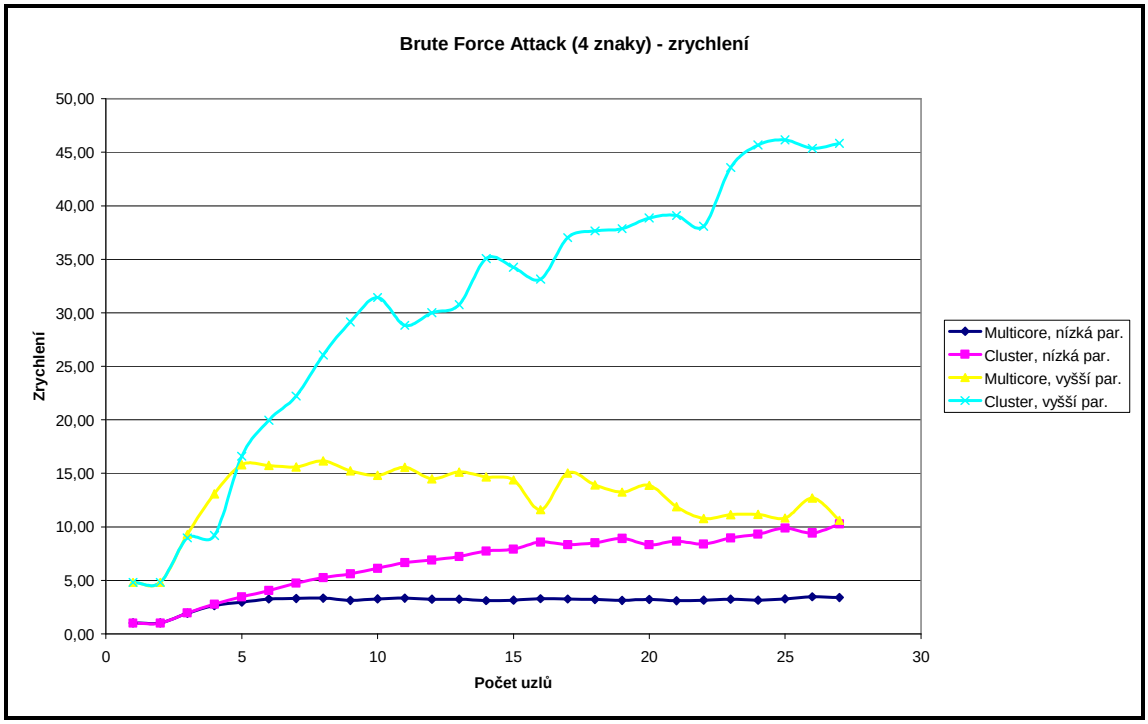
13	41,07	3,24	8,79	15,12	1010,43	2,67	945,27	2,86
14	42,65	3,12	9,06	14,67	1016,87	2,66	1003,08	2,69
15	42,04	3,16	9,24	14,39	1007,11	2,68	970,65	2,78
16	40,30	3,30	11,46	11,60	1074,91	2,51	950,36	2,84
17	40,69	3,27	8,84	15,04	1029,84	2,62	1012,98	2,67
18	41,23	3,22	9,55	13,92	1013,39	2,67	968,37	2,79
19	42,35	3,14	10,03	13,25	1003,92	2,69	1009,69	2,68
20	41,13	3,23	9,57	13,89	1008,48	2,68	1042,98	2,59
21	42,81	3,11	11,18	11,89	1014,30	2,66	965,84	2,80
22	42,13	3,16	12,33	10,78	982,64	2,75	982,31	2,75
23	40,99	3,24	11,92	11,15	1000,71	2,70	996,32	2,71
24	42,17	3,15	11,91	11,16	1008,71	2,68	954,68	2,83
25	40,61	3,27	12,26	10,84	1064,48	2,54	1001,38	2,70
26	38,37	3,46	10,48	12,68	1003,31	2,69	973,73	2,77
27	39,02	3,41	12,52	10,62	1004,72	2,69	968,39	2,79
	jednoduché heslo: Xj5R				**složitější heslo: puPi6**			
	nižší paralelizace		**vyšší paralelizace**		**nižší paralelizace**		**vyšší paralelizace**	
	čas[s]	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**
Cluster								
1	129,75	1,02	27,55	4,83	2703,27	1,00	3348,33	0,81
2	129,75	1,02	27,55	4,83	2703,27	1,00	3348,33	0,81
3	67,72	1,96	14,81	8,98	1564,30	1,73	1704,67	1,58
4	47,74	2,78	14,45	9,20	1137,89	2,37	1189,19	2,27
5	38,39	3,46	8,01	16,60	1024,90	2,64	940,89	2,87
6	32,78	4,06	6,66	19,96	872,26	3,10	810,90	3,33
7	28,02	4,74	5,98	22,23	796,17	3,39	701,76	3,85
8	25,27	5,26	5,10	26,07	762,62	3,54	602,02	4,49
9	23,66	5,62	4,56	29,15	726,17	3,72	572,16	4,72
10	21,65	6,14	4,23	31,43	655,96	4,12	534,43	5,06
11	19,96	6,66	4,61	28,84	647,66	4,17	507,27	5,33
12	19,26	6,90	4,43	30,01	631,18	4,28	502,12	5,38
13	18,36	7,24	4,32	30,77	630,17	4,29	463,39	5,83
14	17,16	7,75	3,79	35,07	580,23	4,66	465,66	5,80
15	16,77	7,93	3,88	34,26	555,44	4,86	441,67	6,12
16	15,49	8,58	4,01	33,15	548,96	4,92	402,02	6,72
17	15,93	8,34	3,59	37,03	564,60	4,79	389,92	6,93
18	15,59	8,53	3,53	37,66	550,32	4,91	397,58	6,80
19	14,91	8,92	3,51	37,87	540,54	5,00	392,27	6,89
20	15,96	8,33	3,42	38,87	538,22	5,02	380,56	7,10
21	15,33	8,67	3,40	39,10	515,03	5,25	375,36	7,20
22	15,84	8,39	3,49	38,09	507,31	5,33	360,05	7,50
23	14,81	8,98	3,05	43,58	520,26	5,19	332,77	8,12
24	14,24	9,34	2,91	45,68	514,20	5,25	349,45	7,73
25	13,45	9,88	2,88	46,16	515,08	5,25	335,23	8,06
26	14,09	9,43	2,93	45,37	512,58	5,27	339,88	7,95
27	12,94	10,27	2,90	45,84	508,01	5,32	333,69	8,10

Tab. 10.4: Naměřené hodnoty, úloha násobení matic

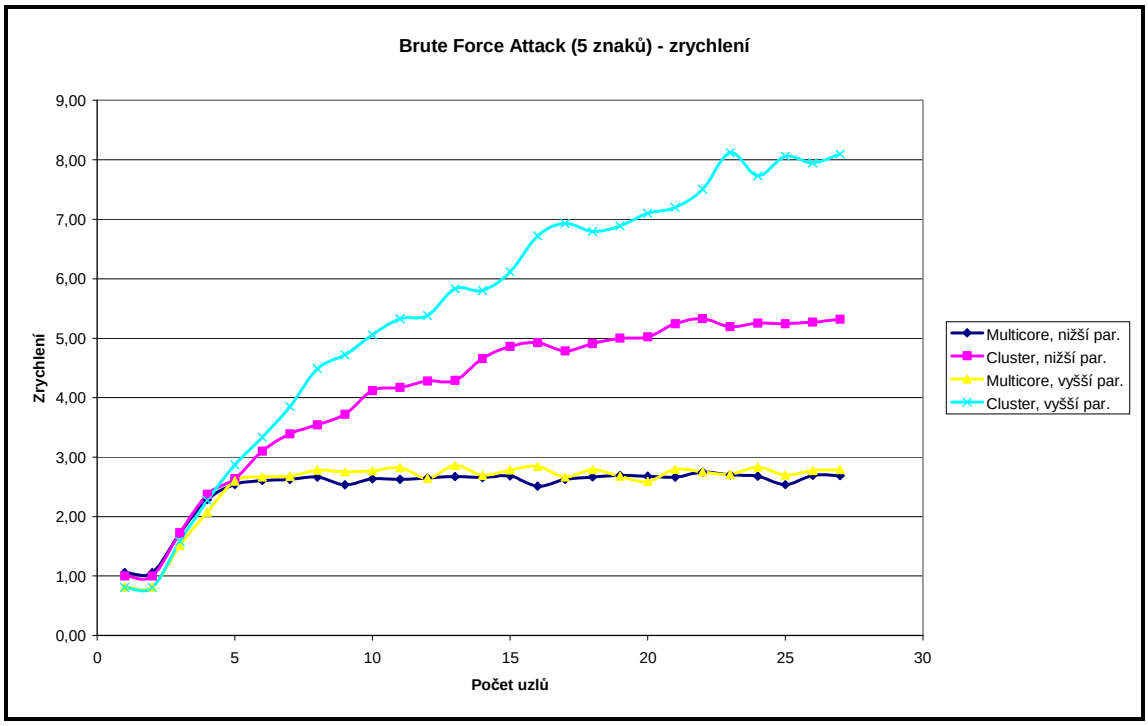
Kvůli lepší přehlednosti byly z naměřených hodnot vytvořeny grafy. Z nich je patrné, že u varianty hledání hesla o čtyřech znacích je potvrzen předpoklad o zrychlení

dle Amdahlova zákona. Se zvyšujícím se počtem procesorů se křivka zrychlení blíží k očekávaným hodnotám. U režimu Multicore se zrychlení projevuje pouze při počátečním přidávání výpočetních jednotek, při použití více než pěti se zrychlení již neprojevuje. U varianty s vyšší paralelizací dokonce mírně klesá. To je způsobeno stylem práce režimu Multicore. Pracuje se sdílenou pamětí, kdy režie na obsluhu paměti nedovoluje vyšší zrychlení aplikace.

U varianty pro hledání hesla o pěti znacích se však výsledky rozcházejí s očekávanými hodnotami. U úlohy řešené v režimu Cluster se podařilo dosáhnout pouze osminásobného zrychlení oproti sekvenční variantě programu. To však neznamená, že vypočítané zrychlení není dosažitelné. Na univerzitním clusteru Hydra jsme omezeni počtem stanic. Pokud bychom přidali další výpočetní jednotky, zrychlení by mělo ještě růst. Dá se však předpokládat, že neporoste až k očekávané hodnotě téměř 55-násobného zrychlení. Je to způsobeno nutností ukládat vygenerovaná hesla do souborů. Jednotlivé uzly se musí dělit o úložiště, což prodlužuje čas potřebný k vykonání programu.



Graf 10.1: Naměřené zrychlení úlohy Brute Force Attack (heslo o délce 4 znaky)



Graf 10.2: Naměřené zrychlení úlohy Brute Force Attack (heslo o délce 5 znaků)

10.2.2 Násobení matic

Úloha maticového násobení byla testována ve dvou základních variantách, lišících se rozměrem vstupních dat. U první varianty měly vstupní čtvercové matice určené k násobení rozměr N = 2000. U druhé varianty, výpočetně náročnější, bylo N = 3000.

Aby bylo vůbec možné porovnávat efektivitu a zrychlení po paralelizaci, musel se nejprve vytvořit sekvenční algoritmus a změřit doba jeho vykonávání.

Jak sekvenční, tak i paralelizovaná verze pracují na podobném principu. Vstupní matice jsou načteny ze souboru do paměti a poté výpočetně zpracovávány. U sekvenční verze byl měřen celkový čas vykonávání, doba potřebná k načtení vstupních matic a dále čas výpočtu, tedy doba potřebná k nalezení výsledné matice vzešlé z násobení vstupů. Měření byla provedena třikrát a následně zprůměrována.

Všechny výše uvedené časy jsou uvedeny v následující tabulce:

Typ úlohy	Měřený úsek	Čas [s]	%
N = 2000	celý sekvenční program	164,99	100
	násobení matic	153,45	93,01
	načítání matic ze vstupních souborů	3,46	2,10
N = 3000	celý sekvenční program	543,04	100
	násobení matic	506,72	93,31
	načítání matic ze vstupních souborů	12,515	2,30

Tab. 10.5: Podíly časů jednotlivých částí sekvenčního programu

Předpokládané maximální zrychlení dle Amdahlova zákona

Aby mohlo být zjištěno předpokládané maximální zrychlení paralelizované aplikace, užijeme vztah vycházející z Amdahlova zákona:

$$\frac{1}{(1-P)}$$

kde P značí paralelní část programu a N je počet použitých procesních jednotek.

Paralelizované verze maticového násobení byly vytvořeny tak, že v první variantě byla paralelizována pouze část násobení matic, ve druhé ještě navíc i načítání matic ze vstupních souborů.

Z výše uvedeného vztahu bylo tedy určeno maximální zrychlení:

Typ úlohy	Co bylo paralelizováno	Maximální možné zrychlení
N = 2000	pouze výpočet	14,3
	výpočet i načtení vstupů	20,4
N = 3000	pouze výpočet	15,0
	výpočet i načtení vstupů	22,8

Tab. 10.6: Možné zrychlení po paralelizaci

Toto ideální zrychlení bylo porovnáno se skutečnými naměřenými hodnotami:

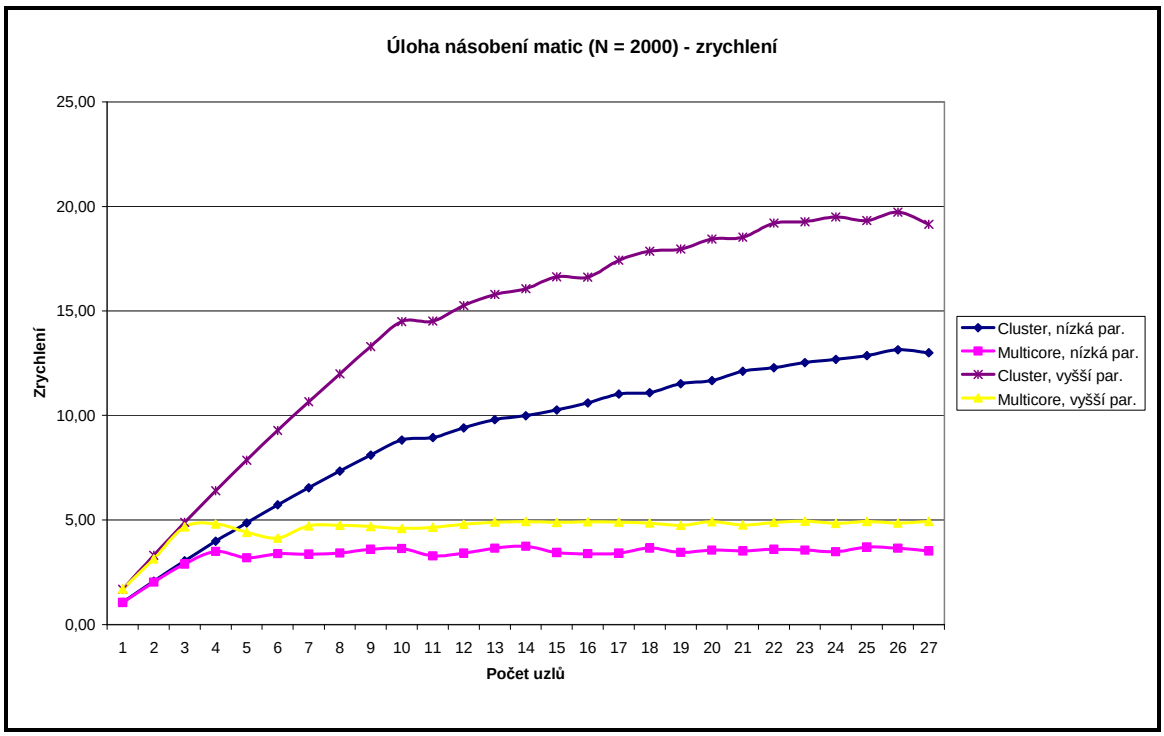
	Matice 2000 * 2000				**Matice 3000 * 3000**			
	nižší paralelizace		**vyšší paralelizace**		**nižší paralelizace**		**vyšší paralelizace**	
	čas[s]	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**
Multicore								
1	156,15	1,06	96,81	1,70	530,78	1,02	343,76	1,58
2	81,17	2,03	52,51	3,14	275,07	1,97	179,78	3,02
3	56,96	2,90	35,28	4,68	186,63	2,91	140,71	3,86
4	47,18	3,50	34,22	4,82	157,14	3,46	110,32	4,92
5	51,64	3,19	37,32	4,42	166,70	3,26	114,36	4,75
6	48,69	3,39	39,84	4,14	156,80	3,46	107,99	5,03
7	49,00	3,37	34,87	4,73	171,22	3,17	109,65	4,95
8	48,22	3,42	34,76	4,75	168,32	3,23	105,36	5,15

9	45,86	3,60	35,13	4,70	169,34	3,21	108,98	4,98
10	45,37	3,64	35,89	4,60	171,95	3,16	110,52	4,91
11	50,15	3,29	35,44	4,66	170,65	3,18	107,13	5,07
12	48,33	3,41	34,39	4,80	168,99	3,21	106,56	5,10
13	45,12	3,66	33,66	4,90	169,43	3,21	112,98	4,81
14	44,05	3,75	33,46	4,93	171,37	3,17	114,20	4,76
15	47,97	3,44	33,73	4,89	170,95	3,18	109,65	4,95
16	48,70	3,39	33,52	4,92	172,69	3,14	112,36	4,83
17	48,30	3,42	33,68	4,90	168,32	3,23	114,94	4,72
18	45,06	3,66	33,96	4,86	174,30	3,12	113,17	4,80
19	47,77	3,45	34,78	4,74	169,99	3,19	109,36	4,97
20	46,28	3,56	33,52	4,92	172,33	3,15	107,90	5,03
21	46,83	3,52	34,65	4,76	170,91	3,18	112,94	4,81
22	45,81	3,60	33,78	4,88	173,33	3,13	113,65	4,78
23	46,35	3,56	33,32	4,95	173,98	3,12	112,85	4,81
24	47,27	3,49	34,03	4,85	174,14	3,12	110,21	4,93
25	44,46	3,71	33,48	4,93	174,95	3,10	109,68	4,95
26	45,15	3,65	33,95	4,86	173,59	3,13	107,60	5,05
27	46,83	3,52	33,48	4,93	174,32	3,12	110,84	4,90
	Matice 2000 * 2000				**Matice 3000 * 3000**			
	nižší paralelizace		**vyšší paralelizace**		**nižší paralelizace**		**vyšší paralelizace**	
Cluster	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**	**čas[s]**	**zrychlení**
1	155,25	1,06	97,85	1,69	531,88	1,02	350,87	1,55
2	79,32	2,08	49,72	3,32	269,29	2,02	176,10	3,08
3	54,01	3,05	33,76	4,89	181,92	2,99	119,16	4,56
4	41,37	3,99	25,75	6,41	138,38	3,92	90,09	6,03
5	33,91	4,86	20,98	7,86	112,36	4,83	73,28	7,41
6	28,79	5,73	17,76	9,29	94,84	5,73	61,47	8,83
7	25,19	6,55	15,48	10,66	82,10	6,61	53,33	10,18
8	22,47	7,34	13,76	11,99	72,77	7,46	47,25	11,49
9	20,35	8,11	12,40	13,31	65,57	8,28	42,31	12,83
10	18,68	8,83	11,38	14,50	59,86	9,07	38,65	14,05
11	18,44	8,95	11,37	14,51	58,64	9,26	37,49	14,48
12	17,52	9,42	10,81	15,26	56,35	9,64	37,10	14,64
13	16,84	9,80	10,45	15,79	53,88	10,08	34,91	15,56
14	16,51	10,00	10,27	16,06	52,07	10,43	33,59	16,17
15	16,06	10,27	9,92	16,64	50,11	10,84	31,86	17,04
16	15,55	10,61	9,93	16,62	47,91	11,33	30,19	17,99
17	14,97	11,02	9,47	17,42	46,42	11,70	29,51	18,40
18	14,87	11,10	9,24	17,86	44,98	12,07	28,92	18,78
19	14,32	11,52	9,19	17,95	44,26	12,27	28,64	18,96
20	14,14	11,67	8,95	18,44	42,65	12,73	28,86	18,82
21	13,62	12,11	8,91	18,52	41,18	13,19	27,21	19,96
22	13,42	12,29	8,60	19,19	40,11	13,54	26,05	20,85
23	13,16	12,53	8,56	19,27	38,77	14,01	26,03	20,86
24	13,01	12,68	8,46	19,49	38,43	14,13	25,65	21,17
25	12,82	12,87	8,54	19,33	38,05	14,27	25,81	21,04
26	12,55	13,15	8,36	19,73	37,96	14,30	25,79	21,06
27	12,69	13,00	8,62	19,14	37,77	14,38	25,80	21,05

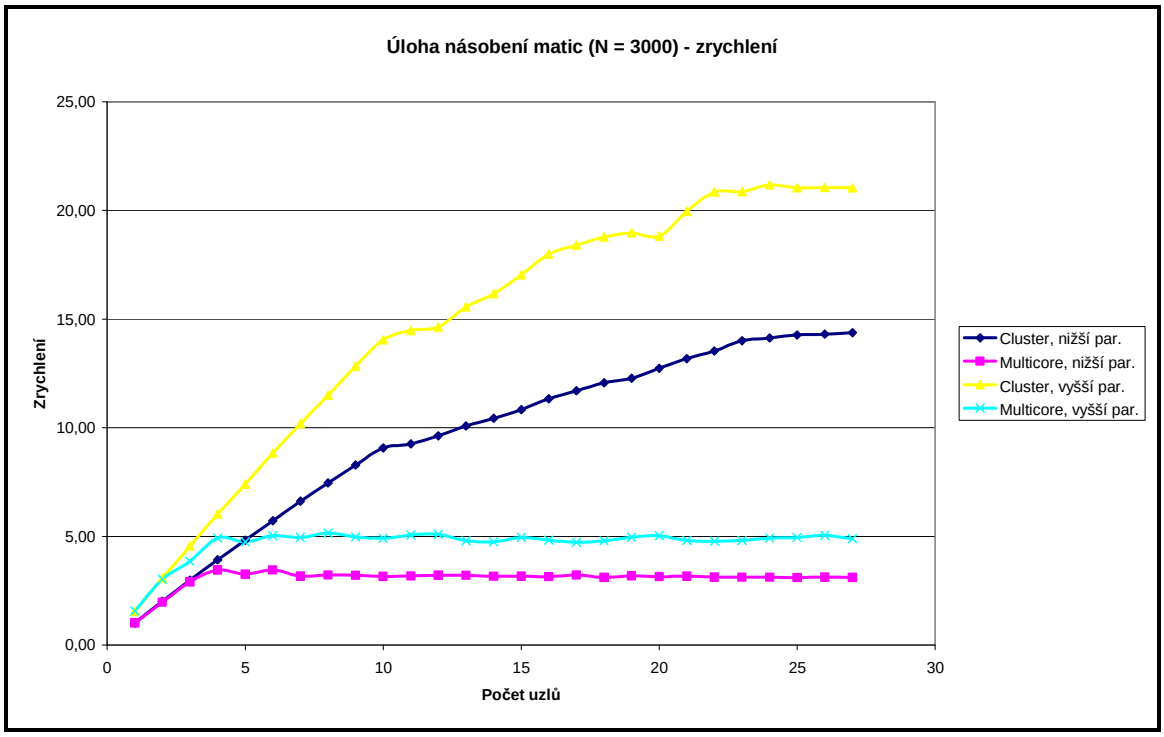
Tab. 10.7: Naměřené hodnoty, úloha násobení matic

Pro přehlednost byly z naměřených hodnot vytvořeny grafy. Z nich je patrné, že se zvyšujícím se počtem procesorů se naměřené zrychlení skutečně přibližuje k vypočteným hodnotám. Přesné vypočítané hodnoty však dosaženo není ani v jednom případě. Jednak je to dáno relativně nízkým počtem výpočetních stanic a také tím, že hodnota je pouze teoretická. Musí být zohledněna režie paralelních algoritmů, která způsobuje zdržení v podobě čekání na systémové zdroje. Dále tuto skutečnost může ovlivňovat nerovnoměrnost zatížení jednotlivých uzlů. Z grafů je patrné, že přidávání dalších uzlů by již nemělo výrazný efekt na rychlost aplikace, jelikož ve všech případech již bylo téměř dosaženo maximálního zrychlení.

U režimu Multicore zdaleka nebylo dosaženo očekávaného efektu zrychlení. Je to zapříčiněno tím, že v módu Multicore pracuje program se sdílenou pamětí. Režie potřebná ke správě paměti způsobí, že zrychlení programu je patrné pouze při přidání několika málo procesních jednotek – jader. Další přidávání je pak neefektivní, zrychlení se neprojevuje.



Graf 10.3: Naměřené zrychlení úlohy násobení matic (N = 2000)



Graf 10.4: Naměřené zrychlení úlohy násobení matic (N = 3000)

10.3 Zhodnocení výsledků

Naměřené výsledky zčásti potvrdily platnost Amdahlova zákona. Pouze u jedné varianty úlohy (Brute Force Attack s vyšší paralelizací) se nepodařilo dosáhnout očekávaných hodnot. Tento výsledek ovšem nelze interpretovat jako neplatnost Amdahlova zákona. Výsledek je ovlivněn použitím společného úložiště souborů, o něž se všechny procesy dělí a přistupují k němu. U režimu Multicore není maximální zrychlení dosažitelné z toho důvodu, že je prováděn na jednom stroji se sdílenou pamětí, tento stroj je dvouprocesorová stanice, kde každý procesor má čtyři jádra.

Lze konstatovat, že výsledky potvrzují platnost Amdahlova zákona.

Závěr

Cílem diplomové práce bylo seznámit se s problematikou paralelního programování na clusterech se speciálním zaměřením na knihovny MPI, MPICH a PVM. Byla kompletně popsána historie vývoje paralelního programování, motivace k tvorbě paralelních algoritmů a standardy definované v této oblasti.

Zvláštní pozornost byla věnována programovacímu jazyka Java, který je v současné době považován za jeden z nejlepších nástrojů pro tvorbu paralelních aplikací. Byly popsány zásadní výhody Javy jako nezávislost na platformě a přenositelnost. Nesporné jsou přednosti v oblasti bezpečnosti a jejího vybavení pro paralelní programování. Podrobně rozebrány byly současné implementace standardu MPI pro Javu, včetně software MPJ Express, který je nainstalován na univerzitním clusteru Hydra.

Autor se také v teoretické části seznámil s problematikou clusterů, s jejich účelem a jednotlivými typy. Specifikovány jsou vlastnosti univerzitního clusteru Hydra, který provozuje Fakulta mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci.

V praktické části byla v souladu se zadáním vytvořena sada srovnávacích testů pro dvě paralelizované úlohy. Jedná se o aplikace pro hledání hesla hrubou silou a pro násobení dvou matic. Aplikace byly vytvořeny za použití knihoven MPI. Testy srovnávají běh aplikací při různé paralelizaci, v různých režimech MPJ Express a při odlišném rozměru vstupních dat – liší se velikost vstupních matic, resp. hledaného hesla.

Při srovnávání výsledků měřených testů s teoretickými hodnotami vycházejícími z Amdahlova zákona bylo zjištěno, že většina testů potvrzuje platnost Amdahlova zákona. Pouze u jednoho testu (Brute Force Attack, varianta s vyšší paralelizací) se nepotvrdil původní předpoklad, důvodem je však způsob práce paralelního algoritmu, který pracuje s datovým úložištěm, což prodlužuje dobu vykonávání programu. V kontextu celé sady srovnávacích testů byla tedy platnost Amdahlova zákona ověřena.

Literatura

[1] BARNAT, Jiří. Analytický model paralelních programů, dostupné z URL: http://www.fi.muni.cz/~xbarnat/IB109/2007-jaro/IB109_05_analyza.pdf

[2] DARWIN, Ian F. Java – Kuchařka programátora. Computer Press, a.s., Brno 2006

[3] GAJDOŠ, Daniel. PVM rozhraní pro DiVinE, Masarykova univerzita v Brně, 2005

[4] KEOGH, James. Java bez předchozích znalostí, Průvodce pro samouky. CP Books, a.s., Brno 2005

[5] MORIN, Steven Raymond. Implementing the Message Passing Interface standard in Java, University of Massachusetts, 2000

[6] PECINOVSKÝ, Rudolf. Myslíme objektově v jazyku Java, 2. vydání, Grada Publishing, Praha 2009

[7] Stručný přehled vlastností a funkcí MPI, dostupné z URL: http://www.nti.tul.cz/wiki/images/b/bb/MPI.pdf

[8] The Source for Java Developers, dostupné z URL: http://java.sun.com

[9] Oficiální stránky projektu MPJ Express, dostupné z URL: http://mpj-express.org/guides.html

Přílohy

Na přiloženém CD se nachází:

1) Zdrojové kódy
   a) Brute Force Attack – varianta s nižší paralelizací
   b) Brute Force Attack – varianta s vyšší paralelizací
   c) Brute Force Attack – sekvenční varianta
   d) Násobení matic – varianta s nižší paralelizací
   e) Násobení matic – varianta s vyšší paralelizací
   f) Násobení matic – sekvenční varianta

2) Vstupní soubory
   a) Vstupní matice o rozměru N = 2000
   b) Vstupní matice o rozměru N = 3000

3) Skript pro zjištění spuštěných úloh na clusteru Hydra

4) Elektronická verze diplomové práce – ve formátech .doc, .rtf, .pdf