Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Výkonové testy v Javě v paralelním a distribuovaném prostředí

Stod 2014 Martin Berka

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

Ve Stodě dne 25. června 2014

Martin Berka

Poděkování

Rád bych tímto poděkoval svému vedoucímu bakalářské práce Ing. Tomáši Potužákovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Abstract

This thesis focuses on efficiency of various algorithms implemented in Java language, used in parallel and distributed environment. The goal of this thesis is to get familiar with standard means of Java language for use in mentioned environments and to create a Java application, which would enable its user to perform various tests in these environments in a uniform way, using several distinct algorithms and communication protocols. Last section of this thesis contains comparison of mentioned algorithms and communication protocols in parallel and distributed environment based on a series of tests performed in created application.

Abstrakt

Práce se zabývá problematikou výkonnosti různých algoritmů implementovaných v jazyce Java při jejich použití v paralelním a distribuovaném prostředí. Cílem této práce je především obeznámení se se standardními prostředky tohoto jazyka pro použití ve zmíněných prostředích a tvorba aplikace v jazyce Java, umožňující jednotným způsobem provádět testy v těchto prostředích, za použití několika vybraných algoritmů a komunikačních protokolů. V závěru práce je dále provedena sada několika testů za pomoci vytvořeného systému a na základě výsledků těchto testů jsou porovnány implementované algoritmy a protokoly z hlediska výhodnosti použití ve zmíněných prostředích.

Obsah

1 Úvod

Práce se zabývá problematikou výkonnosti různých algoritmů implementovaných v jazyce Java při jejich použití v paralelním a distribuovaném prostředí. Cílem práce je především obeznámení se se standardními prostředky tohoto jazyka pro použití v daných prostředích. Následně by měla být vytvořena aplikace v tomto jazyce, umožňující pomocí uživatelsky přívětivého prostředí jednotným způsobem vzájemně porovnat výkon několika vybraných algoritmů v daných prostředích, s využitím zmíněných prostředků a vybraných protokolů.

Aplikace by měla za pomoci jednoduchého grafického rozhraní umožňovat volbu mezi paralelním a distribuovaným prostředím, dále volbu jednoho z implementovaných algoritmů pro použití během testu a v případě volby distribuovaného prostředí také volbu protokolu použitého pro komunikaci v distribuovaném prostředí a zda má aplikace sloužit jako řídicí proces (farmer), tj. proces rozdávající úkoly dalším procesům a přijímající výsledky, či pracovní proces (worker), tj. proces provádějící samotné výpočty. Aplikace by vzhledem ke své povaze měla také zobrazovat vhodné statistické údaje, jako např. časy potřebné pro výpočty jednotlivých úkolů, celkový čas a množství dat přenesených po síti.

Dále by výsledkem práce měl být soubor výsledků testů provedených ve vytvořené aplikaci, popisující rozdíly ve výkonnosti a vhodnost použití jednotlivých protokolů a algoritmů ve zmíněných prostředích.

2 Konkurentní programování v Javě

Značnou výhodou jazyka Java je skutečnost, že již obsahuje mnoho prostředků pro programování v paralelním či distribuovaném prostředí. Nejpoužívanější prostředky pro tato prostředí jsou již implementována a programátor tak může budovat aplikace nad těmito existujícími implementacemi[1].

2.1 Paralelní prostředí

Při provádění výpočtů v paralelním prostředí se značně zjednodušuje problém výměny dat mezi procesorovými jádry provádějící výpočet, neboť se ve většině případů pracuje se sdílenou pamětí. Pravděpodobně největším problémem je tedy v paralelním prostředí ošetření kritických sekcí a zajištění správného pořadí operací jednotlivých kroků, neboli synchronizace jednotlivých prvků – nejčastěji vláken – programu provádějícího výpočet.

Java poskytuje pro synchronizaci značné množství prostředků, z nichž většinu nalezneme v balíku `java.util.concurrent`[2]. Zde uvádím ty nejčastěji používané z nich.

2.1.1 Zámek

Zámky patří mezi základní a nejjednodušší synchronizační primitiva, ze kterých většinou vycházejí všechna ostatní. V jazyce Java jsou zámky relativně komplexní a nabízejí širokou funkcionalitu. Pokud je zámek odemčen, vlákno může daný zámek uzamknout a pokračovat v práci, v opačném případě je vlákno pozastaveno do doby, než je zámek jiným vláknem odemčen. Balík `java.util.concurrent.locks` poskytuje několik typů zámků, z nichž nejpoužívanější jsou především `ReentrantLock` a `ReentrantReadWriteLock`[3].

`ReentrantLock`, jak již plyne z názvu, je reentrantní zámek (tedy umožňující vícenásobný přístup). Zámek smí být uzamknut vždy pouze jedním vláknem (voláním metody `lock()`), avšak toto vlákno ho může uzamknout

dle potřeby i vícekrát – toto zajišťuje, že vlákno nemůže „samo sobě“ způsobit uvíznutí (deadlock). Na uvolnění zámku je však nutné, aby dané vlákno provedlo také stejný počet odemknutí (metodou `unlock()`), jinak nemůže být zámek uzamknut jiným vláknem[3].

`ReentrantReadWriteLock` je dvojicí reentrantních zámků, určených především pro ošetření přístupu k datům, která jsou často čtena a méně častěji měněna či zapisována. Zámek dovoluje vícenásobný přístup pro čtení dat, ale exkluzivní přístup pro jejich zápis – čili neomezené množství vláken může číst data zároveň (v takovém případě nemůže dojít k žádnému problému), ale pokud chce některé vlákno data zapisovat, nesmí v tu chvíli jiné vlákno držet zámek pro čtení a pouze dané vlákno smí jako jediné držet zámek pro zápis[4].

Zajímavou možností u těchto zámků je možnost nastavení `fairness`, neboli férovosti, spravedlivosti. Pokud by o zámek velmi často usilovalo velké množství vláken, mohlo by dojít k situaci, kdy se některému z vláken získání daného zámku dlouho nebo vůbec nepodaří (zámek mu vždy sebere nějaké jiné vlákno). V případě nastavení zámku jako spravedlivého či férového je zajištěno mnohem spravedlivější rozdělení zámku mezi jednotlivá vlákna – avšak za cenu mnohem větší režie. Záleží tedy, zda programátor chce spravedlivý přístup pro všechna vlákna za cenu větší režie, či vyšší výkon za menší spravedlnost[5].

2.1.2 Semafor

Semafory poskytují obdobnou funkci jako základní zámky, avšak do kritické sekce chráněné semaforem může vstoupit i více vláken. Obecně semafor sestává z čítače „povolení“ a dvou metod, nejčastěji označované jako `V()` a `P()` či `signal()` a `wait()`, které slouží ke zvyšování a snižování hodnoty zmíněného čítače. Vlákno, které dekrementací sníží čítač na zápornou hodnotu je povinno se zablokovat a čekat, dokud není čítač jiným vláknem opět navýšen[6].

V Javě jsou semafory implementovány pomocí třídy `java.util.concurrent.Semaphore`. Místo zmíněných obecných názvů me-

tod pro práci s čítačem povolení obsahuje tato třída metody `acquire()` (snížení čítače) a `release()` (navýšení čítače). Kromě těchto metod poskytuje tato třída množství dalších metod, například pro zjišťování současného stavu čítače (metoda `availablePermits()`), počtu vláken zablokovaných a čekajících na povolení (metoda `getQueueLength()`), či zjištění „spravedlivosti“, dříve zmíněné v části 2.1.1[7].

2.1.3 Monitor

Monitor je speciální konstrukcí programovacího jazyka a jedná se o vysokoúrovňové synchronizační primitivum – z hlediska samotného programátora je značně jednodušší na použití než zámky a semafory (se kterými je ale prakticky ekvivalentní) a snižuje tak možnost udělat nějakou závažnou chybu. Veškerá data tvořící kritickou sekci jsou dostupná pouze z vnitřku monitoru a vstup do monitoru je chráněn klasickým zámkem – vlákno tedy musí nejprve získat daný zámek monitoru a následně do něj může vstoupit a pracovat nad jeho daty. Po opuštění monitoru nebo při zablokování se nad nějakou podmínkovou proměnnou musí samozřejmě zámek opět uvolnit[6].

Zajímavostí je v Javě skutečnost, že každý objekt má se sebou spjat svůj vlastní monitor – proto má každý objekt v Javě metody `wait()`, `notify()` a `notifyAll()`, kde metoda `wait()` způsobí zablokování se nad podmínkovou proměnnou daného monitoru, metoda `notify()` probudí první z čekajících vláken a metoda `notifyAll()` probudí všechna čekající vlákna tohoto monitoru[8].

Další zajímavou možností je označení metody klíčovým slovem `synchronized` v její hlavičce – tím označíme metodu za patřící do monitoru objektu, jehož je metoda součástí. Programátor se dále již nemusí o nic dalšího starat, metodu daného objektu bude smět vykonávat v jednu chvíli vždy pouze jediné vlákno. Podobně je možné „obalovat“ ne pouze celé metody, ale i menší bloky kódu konstrukcí `synchronized(objekt){...}`, kde `objekt` označuje objekt, do jehož monitoru bude daný kus kódu spadat. Monitory byly v Javě dlouhou dobu jediným synchronizačním primitivem, než byl přidán již dříve zmiňovaný balíček `java.util.concurrent`.

2.1.4 Bariéra

Bariéry se využívají pro synchronizaci dvou a více vláken – v praxi je toto potřeba například pokud na výpočtu pracuje samostatně několik vláken provádějících dílčí kroky a pro provedení dalšího kroku jsou potřeba výsledky z několika nebo všech těchto vláken. Při použití bariéry se při jejím vytvoření nastaví počáteční čítač podobně jako u semaforu (nebot' bariéra je jakýmsi „inverzním semaforem“ a jakékoli vlákno které „zastaví“ na této bariéře je pozastaveno, dokud na ní nezastaví předem nastavený počet vláken, které se mají na bariéře synchronizovat. Ve chvíli, kdy je čítač bariéry snížen na nulovou hodnotu, je tato odstraněna a všechna vlákna čekající na této bariéře jsou opět spuštěna. Výhodou některých bariér oproti semaforům je fakt, že vlákno může snižovat jejich čítač aniž by u nich zastavilo – může rovnou pokračovat v práci[6].

V Javě se jedná především o třídy `CountDownLatch` a `CyclicBarrier`, z nichž první funguje jako „jednorázová“ bariéra kterou lze použít pouze jednou a druhá (jak již plyne z názvu) jako cyklická bariéra, kterou lze používat opakovaně[9].

2.2 Distribuované prostředí

Distribuovaným prostředím budu pro účely této práce považovat dva a více počítačů, spojených počítačovou sítí za účelem spolupráce na nějakém zpracování dat, nebot' definice distribuovaného prostředí se v různých publikacích liší dle konkrétní oblasti použití.

Pokud si uvědomíme, že jednotlivé počítače distribuovaného systému jsou vlastně obdobou samostatných procesorů jednoho počítače, je jasné, že budeme potřebovat minimálně některé synchronizační prostředky zmíněné v předchozí části. Oproti práci v „pouze“ paralelním prostředí je tu však problém komunikace – vlákna jednoho počítače v naprosté většině případů sdílejí určitou část paměti, kterou lze snadno využít pro předávání dat mezi nimi. V distribuovaném prostředí je však potřeba zajistit výměnu dat (obecně komunikaci) za pomoci počítačové sítě. Stejně jako prostředky pro paralelní prostředí nabízí Java ve svých základních knihovnách několik prostředků pro

běžnou komunikaci počítačů po síti. Opět uvedu a popíšu jen ty nejčastěji a běžně používané.

2.2.1 Transmission Control Protocol

Transmission Control Protocol (dále jen TCP) je zástupcem rodiny TCP/IP protokolů a v dnešní době jedním z nejpoužívanějších protokolů pro přenos dat vůbec. TCP je spojově orientovaný, zaručuje spolehlivé doručování a doručování ve správném pořadí. Nevýhodou oproti tomu je větší režie a tím pádem větší zatížení sítě než např. u protokolu UDP (viz dále)[10].

Java pro použití protokolu TCP nabízí množství prostředků, převážně v balících `java.net` a `java.io` (pro vstupní a výstupní proudy). Základem je použití třídy `ServerSocket`, která slouží k vytvoření serverového soketu a následné práci s ním. Soket je přiřazen na zadanou adresu a port kde naslouchá a čeká na připojení klientů – k tomu slouží třída `Socket`, která naopak slouží k připojení se k naslouchajícímu serveru[11].

Protože TCP protokol slouží pro přenos toku bajtů, využívá datových proudů (streamů). Zmíněné sokety obsahují metody pro získání referencí na vstupní a výstupní proudy, do kterých lze zapisovat a ze kterých lze číst proudy bajtů. K tomuto účelu je tedy nutné, aby cokoli, co má být přeneseno, mohlo být převedeno na sekvenci bajtů – neboli serializováno. Toho lze v Javě docílit snadno pouhou implementací rozhraní `Serializable`, což je ve skutečnosti prázdné rozhraní sloužící pouze jako značka (marker). Java se o automatickou serializaci a opětovnou deserializaci na druhé straně umí postarat sama, takže programátor pro to nemusí kromě implementace zmíněného rozhraní nic dělat a může do vstupního proudu soketu přímo zapisovat libovolný objekt splňující toto rozhraní (a taktéž ho z výstupního proudu číst) s využitím metod `writeObject()` a `readObject()`[12].

2.2.2 User Datagram Protocol

User Datagram Protocol (dále jen UDP) je druhým zástupcem rodiny protokolů TCP/IP a představuje ve srovnání s TCP jednodušší protokol. Jeho charakteristickým rysem je menší zatížení sítě než při použití protokolu TCP,

avšak také jeho nespolehlivost při doručování jednotlivých paketů – UDP nezaručuje, že se během přenosu žádná data neztratí či nepřijdou v nesprávném pořadí. Tyto rysy jsou v mnoha případech přípustné (např. přenos hlasu či videa v reálném čase, kdy se spokojíme se ztrátou některých dat za cenu nižší latence), avšak pro výpočty v distribuovaném prostředí jsou vhodnější spíše protokoly, které zaručí bezchybný přenos. Java pro protokol UDP opět nabízí prostředky především prostřednictvím balíku `java.net`, v němž nejpodstatnější jsou třídy `java.net.DatagramSocket` a `java.net.DatagramPacket`[10].

2.2.3 New I/O

New I/O (dále jen NIO) je soubor nástrojů jazyka Java pro intenzivní vstupně-výstupní (I/O) operace, dostupný od verze Java 1.4 (balík `java.nio`). Ve verzi 1.7 bylo dále přidáno rozšíření NIO2, poskytující nové API pro práci se soubory. NIO umí pracovat nad oběma dříve zmíněnými protokoly[13, 14].

Cílem Java NIO je poskytovat jednotné rozhraní pro používání proudů různých typů, bez ohledu na konkrétní zdroj či cíl (pevný disk, síť, ...), které se následně snaží docílit konkrétních operací čtení a zápisu využitím co nejefektivnějších (nízkoúrovňových) operací dané platformy. Na rozdíl od „klasického“ I/O, ve kterém pracujeme s proudy bajtů (viz předchozí TCP), NIO využívá principu kanálů (`channels`) a bufferů (`buffers`)[13, 14].

Kanály fungují obdobně jako dříve zmíněné proudy (streamy), s tím rozdílem, že:

- do kanálů lze jak zapisovat, tak z nich číst, zatímco proudy jsou většinou jen pro jeden z těchto účelů
- do kanálů lze zapisovat a číst z nich asynchronně
- do kanálů lze zapisovat nebo z nich číst vždy pouze s použitím bufferu

Java NIO nabízí několik implementací kanálů, z nichž nejdůležitějšími jsou především `FileChannel` (práce se soubory), `DatagramChannel` (práce s datagramy – tedy především UDP protokolem), `SocketChannel` a `ServerSocketChannel` (práce především s TCP protokolem)[13].

Buffery jsou prakticky paměťovými bloky pro zápis a čtení dat, zabalené do objektu poskytující soubor metod pro jednodušší manipulaci. Jak již bylo zmíněno, do bufferů jsou načítána data ze zmíněných kanálů a obdobně jsou z nich do kanálů opět zapisována. Typická práce s bufferem zahrnuje zápis dat do bufferu, zavolání metody `flip()`, která připraví buffer pro čtení, přečtení dat z bufferu a následné vyprázdnění dat metodami `clear()` (odstranění veškerých dat v bufferu) nebo `compact()` (odstranění pouze dat, která již byla přečtena a posun zbývajících dat na začátek bufferu)[15]. NIO poskytuje buffery pro prakticky veškeré primitivní datové typy, vzhledem k povaze této práce se budu zabývat především buffery typu `ByteBuffer` pro ukládání a čtení obecných sekvencí bajtů.

Selektory (`selectors`) jsou další zajímavou komponentou Java NIO, která si rozhodně zaslouží zmínku. Selektor je komponenta, která umožňuje sledování a následnou obsluhu jednoho nebo více kanálů s využitím pouze jediného vlákna programu. Jakýkoli existující kanál lze zaregistrovat do selektoru. Ten následně tento kanál periodicky monitoruje a programátor si může sám nadefinovat, co se provede v případě, že je libovolný z monitorovaných kanálů nově připojený, připravený k připojení či zápisu dat nebo jejich čtení. Tímto způsobem lze ušetřit režii operačního systému při neustálém přepínání vláken v případě, že víme že operace nad danými kanály nejsou natolik intenzivní a zdlouhavé, aby bylo nutné mít pro obsluhu každého z nich vlastní vlákno[16].

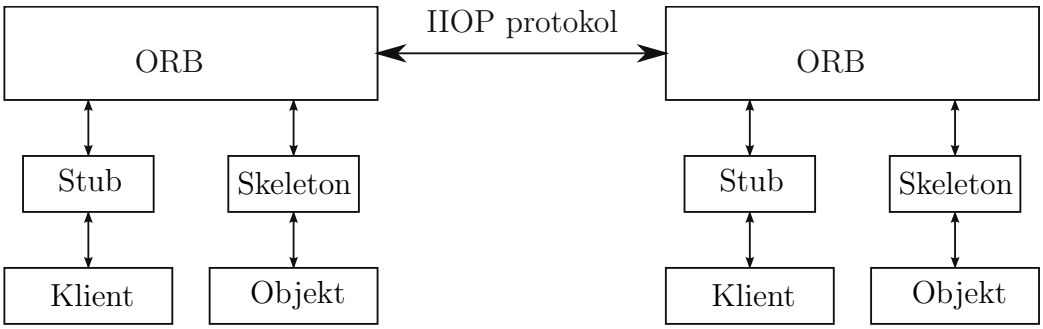
Kromě zde zmíněných prostředků nabízí NIO samozřejmě množství dalších, zde neuvedených prostředků, jako například `Pipe`, `FileLock` apod., nicméně se jedná většinou o pomocné utility pro použití se zmíněnými třemi základními komponentami[13].

2.2.4 Common Object Request Broker Architecture

Common Object Request Broker Architecture (dále jen CORBA) je standardem, umožňujícím spolupráci systémů běžících na různých platformách, operačních systémech a využívající různé programovací jazyky. Cíle tohoto standardu jsou obdobné jako u objektově orientovaného programování – zapouzdření a opakované využití. Ačkoli CORBA využívá objektově orientovaný model, samotné systémy využívající tento standard objektově oriento-

vané být nemusí – v takovém případě ale musí programátor některé objektově orientované vlastnosti v daném programu emulovat[17].

Pro specifikaci rozhraní jednotlivých objektů využívá CORBA Interface Definition Language (IDL). Specifikovaná rozhraní jsou následně mapována do konkrétního programovacího jazyka, přičemž existují standardní mapování pro množství programovacích jazyků (např. Ada, C/C++, Java či Python). Samotná komunikace mezi jednotlivými systémy probíhá pomocí Object Request Broker (ORB), což je jakýsi prostředník, který zprostředkovává danou komunikaci pomocí tzv. *marshallingu*, neboli převodu jednotlivých struktur do bajtových proudů vhodných pro přenos po síti a naopak. ORB určitým způsobem skrývá implementační detaily přenosu dat a činí tak vzdálené objekty (objekty na jiných systémech) z hlediska používání objekty lokálními – struktury na lokálním i vzdáleném systému je pak možné využívat naprosto identicky[17].



Obrázek 2.1: Znázornění práce CORBA architektury[17]

2.2.5 Java Remote Method Invocation

Java Remote Method Invocation (dále jen RMI) je vyšším prostředkem jazyka Java, který umožňuje volání metod objektů na jiných počítačích – jeho původní implementace je obdobou zmíněného standardu CORBA pro jazyk Java, jeho novější implementace je pak přímo implementací architektury CORBA[18].

RMI používá pro komunikaci spojení mezi dvěma (Javovskými) virtuálními stroji pomocí protokolu TCP/IP, nad kterým pak používá v původní

implementaci protokolu Java Remote Method Protokol (JRMP) nebo dále volitelně Internet Inter-ORB Protocol (IIOP), který právě umožňuje spolupráci s ne-Javovskými systémy a programy podle CORBA standardu[19].

Průběh komunikace při využití varianty JRMP vypadá následovně[20]:

- Server (většinou počítač, který chce poskytovat své objekty ostatním, obecně však jakýkoli přístupný počítač v síti) spustí RMI registr, který slouží pro ukládání a zpřístupnění referencí na vzdálené objekty serverů klientům.
- Server, který chce poskytovat své objekty klientům, zanese do zmíněného RMI registru referenci na objekt, který chce poskytovat – tzv. *stub*.
- Klient si z RMI registru stáhne příslušný stub – ten obsahuje veškeré potřebné informace pro komunikaci se serverem – a od té chvíle může volat metody, který tento stub nabízí.
- V případě, že by klient chtěl používat metody serveru využívající třídy které sám klient nezná (klientská aplikace nemá definice těchto tříd), může si tyto definice za běhu stáhnout z předem definovaného webového serveru, který definice těchto tříd obsahuje. Toho lze hojně využít v případě, že serverové a klientské aplikace jsou vyvíjeny odděleně – vývojář serverové aplikace pouze dodá definice nově implementovaných tříd na webový server a klientské aplikace si je pak mohou v případě potřeby stáhnout[20].
- Server se při přijetí požadavku postará o rozbalení tzv. *marshall streamu*, čili proudu informací obsahujícím např. parametry předávané volané metodě, vyvolání této metody u konkrétního objektu, zachycení návratové hodnoty či vyvolané výjimky, opětovné zabalení této informace do marshall streamu a následné vrácení klientovi[21].

Toto vše se děje z pohledu programátora transparentně – programátor musí tedy zajistit pouze prvotní export objektů do RMI registru serveru a následné získání referencí na tyto objekty u klienta. Poté již může pracovat se získanou referencí jako s referencí na běžný lokální objekt.

3 Algoritmy pro testování

Aby bylo možné vhodně a objektivně porovnat výkonnost při použití paralelního a distribuovaného prostředí (navíc za použití různých způsobů komunikace), je žádoucí zvolit a implementovat několik různorodých algoritmů. Algoritmy by se měly lišit především v aspektech jako je celková složitost, distribuovatelnost (zda je lze snadno rozdělit na menší problémy, které lze řešit nezávisle) či náročnost na výměnu dat – některé algoritmy by měly např. být méně náročné na výpočet a vyžadovat větší míru komunikace a některé naopak (tj. vyšší náročnost, málo dat).

3.1 Inverze matice

3.1.1 Popis

Inverzní matice k dané matici je taková matice, která po vynásobení s původní maticí dá jednotkovou matici. Inverzní matici lze sestrojit pouze pro matice regulární, tedy čtvercové matice s determinantem různým od nuly. Pro samotný výpočet inverzní matice existuje množství algoritmů, jako např. Gaussova eliminační metoda, LU rozklad, Choleského rozklad apod. Následující příklad ukazuje výpočet inverzní matice za pomoci Gaussovy eliminační metody. Mějme dánu matici A[22].

$$A = \left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \right] \tag{3.1}$$

Postup začneme tím, že napravo od dané matice připíšeme jednotkovou matici. Získáme tak rozšířenou matici A_i.

$$A_i = \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 3 & 4 & 0 & 1 \end{array} \right] \tag{3.2}$$

Nyní tuto celou matici upravujeme pomocí řádkových a sloupcových úprav tak, abychom získali jednotkovou matici na levé straně. Na začátek přičteme ke druhému řádku -3 násobek prvního řádku.

$$A_i = \left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 0 & -2 & -3 & 1 \end{array} \right] \tag{3.3}$$

Nyní přičteme druhý řádek k prvnímu řádku, čímž získáme nulu na pozici a_{12}.

$$A_i = \left[\begin{array}{cc|cc} 1 & 0 & -2 & 1 \\ 0 & -2 & -3 & 1 \end{array}\right] \tag{3.4}$$

Následně vydělíme druhý řádek -2, čímž již získáme na levé straně požadovanou jednotkovou matici.

$$A_i = \left[\begin{array}{cc|cc} 1 & 0 & -2 & 1 \\ 0 & 1 & \frac{3}{2} & -\frac{1}{2} \end{array}\right] \tag{3.5}$$

Na pravé straně matice A_i jsme nyní získali matici I, která je inverzní k původní matici A.

$$I = \left[\begin{array}{cc} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{array}\right] \tag{3.6}$$

3.1.2 Vstup, výstup a distribuovatelnost

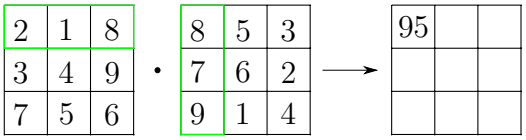
Vstupem i výstupem algoritmu jsou regulární čtvercové matice stejné dimenze – velikost vstupních a výstupních dat je tedy stejná. Jedná se o problém, jehož distribuovatelnost závisí částečně na algoritmu použitém pro samotné nalezení inverzní matice – některé metody vyžadují přenos celé matice, zatímco jiné, pokročilejší metody (např. blokový LU rozklad) jsou pro inverzi matic v distribuovaném prostředí vhodnější, neboť v něm dosahují lepšího výkonu. [23].

3.2 Násobení matic reálných náhodných čísel

3.2.1 Popis

Násobení matic je proces, po kterém je prvek v i-tém řádku a j-tém sloupci výsledné matice výsledek skalárního součinu vektoru i-tého řádku první matice s vektorem j-tého sloupce druhé matice (viz obr. 3.1). Aby bylo možné provést vynásobení dvou matic, musí se počet sloupců první matice rovnat počtu řádků druhé matice[24]. Výpočetní náročnost základního algoritmu pro násobení matic je $\mathrm{O}(n^3)$. Existují však i další algoritmy s nižší výpočetní

složitostí, jejichž použití však často vede k nižší numerické stabilitě (např. Strassenův algoritmus)[25].



Obrázek 3.1: Ukázka výpočtu jednoho prvku výsledné matice při násobení dvou matic

3.2.2 Vstup, výstup a distribuovatelnost

Násobení matic s náhodnými čísly představuje algoritmus, u kterého je poměr velikostí vstupních a výstupních dat závislý na rozměrech násobených matic, neboť výsledná matice má rozměry $A_s \times B_r$, kde A_s je počet sloupců první matice a B_r je počet řádků druhé matice[24]. Obecně lze takovou úlohu snadno distribuovat, nicméně pro výpočet jednoho prvku potřebujeme znát všechny prvky jednoho sloupce první matice a jednoho řádku druhé matice.

3.3 Umocňování matice reálných náhodných čísel

3.3.1 Popis

Umocňování matice je prakticky pouze variantou již zmíněného násobení matic – matici jednoduše opakovaně násobíme sebou samou. Z pravidla pro násobení matic plyne, že daná matice musí být čtvercová – jinak by si neodpovídaly počty řádků a sloupců[24].

3.3.2 Vstup, výstup a distribuovatelnost

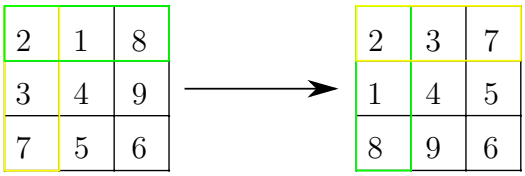
V porovnání s dříve zmíněným algoritmem běžného násobení matic se tento algoritmus však značně liší v závislosti vstupu a doby výpočtu. Pouhé zvýšení

exponentu, které velikost vstupu nijak značně nezvýší, může vyústit v mnohonásobně delší dobu výpočtu. Algoritmus je tedy značně variabilní – můžeme zvolit vstupní data taková, která budou malá a výpočet potrvá značnou dobu (matice 100×100 umocněná exponentem 1000), nebo naopak větší vstupní data s nižší dobou výpočtu (matice 1000×1000 umocněná exponentem 2). Podobně jako u klasického násobení matic je potřeba pro výpočet jednoho prvku matice znát všechny prvky daného sloupce a řádku[24].

3.4 Transpozice matice

3.4.1 Popis

Transpozice matice je zástupcem výpočetně nenáročných algoritmů. Laicky řečeno, transponovaná matice k matici A je taková matice, jejíž řádky jsou sloupce původní matice a naopak – její sloupce jsou řádky původní matice. Výsledná matice má tedy obrácené počty řádek a sloupců oproti matici původní[24].



Obrázek 3.2: Ukázka transpozice matice

3.4.2 Vstup, výstup a distribuovatelnost

Vstupní a výstupní data pro transpozici matice mají vždy identickou velikost – počty prvků ve vstupních a výstupních datech jsou stejné, stejně tak obě matice mají naprosto identickou množinu hodnot svých prvků. Algoritmus je celkem dobře distribuovatelný – každý proces může zpracovávat určitou množinu řádků či sloupců, nezávisle na ostatních.

3.5 Raytracing

3.5.1 Popis

Raytracing je metoda pro vykreslování 3D grafiky obsahující velké množství interakcí mezi světly a povrchy – tedy např. scén s nejrůznějšími světelnými zdroji, zrcadly, průhlednými a lesklými předměty apod. Na rozdíl od „intuitivního“ postupu od světelného zdroje k oku pozorovatele se u této metody používá postup opačný, tedy sledování paprsku od oka pozorovatele (kamery) směrem ke zdroji světla, což je výhodné, neboť se tak neprovádí sledování paprsků, které se k pozorovateli vůbec nedostanou[26].

3.5.2 Vstup, výstup a distribuovatelnost

Raytracing je jednou z metod, kde se poměr mezi vstupními a výstupními daty může a nemusí značně lišit – vše závisí především na požadované velikosti výsledné scény. Při distribuci výpočtu je nutné distribuovat popis celé scény a výsledkem je pak obrázek dané scény.

3.6 Hledání dělitelů

3.6.1 Popis

Hledání dělitelů je úloha, která se snaží o nalezení všech celočíselných dělitelů zadaného čísla (případně pouze do nějaké menší zadané meze). Číslo X je dělitelem čísla Y v případě, že zbytek po vydělení čísla Y číslem X je nula. Nejjednodušším způsobem nalezení všech celočíselných dělitelů je průběžné dělení zadaného čísla všemi čísly od 1 do zadaného čísla a ověřování, zda je zbytek nulový. Takový algoritmus je však zbytečně náročný, neboť stačí dělit pouze všemi čísly od 2 do hodnoty odmocniny zadaného čísla, neboť platí, že každé celé číslo je dělitelné číslem 1 a $\sqrt{x} \cdot \sqrt{x} = x$[27].

3.6.2 Vstup, výstup a distribuovatelnost

Velikost vstupních dat je u zmíněného algoritmu zanedbatelná – jedná se (pro každou úlohu) pouze o jedno číslo, k němuž chceme najít všechny dělitele. Naopak velikost výstupních dat může být různá – číslo nemusí mít žádného dělitele, či může mít naopak velkou řadu dělitelů. Co se týče distribuovatelnosti, jedná se o algoritmus velmi snadno distribuovatelný – každý proces může nezávisle na ostatních hledat dělitele pouze v určitém stanoveném rozsahu.

3.7 Eratosthenovo síto

3.7.1 Popis

Eratosthenovo síto je algoritmus pro nalezení všech prvočísel od čísla 2 do určité zadané meze. Algoritmus pracuje tak, že odebere první číslo v pořadí (tj. na začátku číslo 2) a toto číslo označí za prvočíslo. Následně odstraní ve zbylé posloupnosti všechny násobky daného čísla a celý proces opakuje, dokud nezpracuje všechna čísla do zadané meze[28].

3.7.2 Vstup, výstup a distribuovatelnost

Velikost vstupních dat je u tohoto algoritmu téměř vždy mnohem menší, než velikost dat výstupních – vstupem je pouze daná mez, zatímco výstupem je posloupnost všech prvočísel do zadané meze. Jedná se o algoritmus, který se obtížně distribuuje, protože výpočet dalších prvočísel je závislý na výpočtu v předchozím kroku.

3.8 Prohledávání stavového prostoru

3.8.1 Popis

Prohledávání stavového prostoru není samo o sobě konkrétním algoritmem, ale spíše obecnou metodou řešení úloh, které jsou definovány určitým počátečním stavem, jedním nebo více koncovými stavy a množinou přechodů mezi těmito stavy. Úkolem prohledávání stavového prostoru je pak nalezení takové posloupnosti přechodů mezi stavy, která nám umožní dostat ze z počátečního stavu do jednoho ze žádaných koncových stavů. Vzhledem k tomu, že stavový prostor lze reprezentovat orientovaným grafem se prakticky jedná o algoritmy pro hledání cesty v grafu[29].

3.8.2 Vstup, výstup a distribuovatelnost

Vstupní data pro algoritmy prohledávání stavového prostoru jsou obecně větší, než data výstupní – vstupem je celý stavový prostor, příp. jeho část, zatímco výstupem je pouze posloupnost stavů vedoucí k některému z požadovaných koncových stavů, což jsou ve většině případů data zanedbatelné velikosti. Při distribuci problému prohledávání stavového prostoru záleží na konkrétním zvoleném algoritmu, obecně je nutné přenášet minimálně část stavového prostoru, který chceme prohledávat[29].

3.9 Mergesort

3.9.1 Popis

Mergesort je metodou řazení typu rozděl a panuj s asymptotickou složitostí $\mathrm{O}(n \times log(n))$. Algoritmus pracuje opakovaným rozdělováním vstupních dat do dvou menších samostatných celků, které jsou následně samostatně seřazeny a nakonec sloučeny do finální posloupnosti. Jedná se o stabilní řadicí algoritmus, tedy neměnící pořadí prvků se stejným klíčem (v případě řazení čísel tedy stejnou numerickou hodnotou)[30].

3.9.2 Vstup, výstup a distribuovatelnost

Velikost vstupních a výstupních dat u algoritmu mergesort je identická (neseřazená posloupnost čísel vs. stejně dlouhá, ale seřazená posloupnost). Z jeho podstaty se jedná o typ algoritmu, který je velmi vhodný pro distribuovaný výpočet. Při distribuci je nutné přenášet pouze část posloupnosti, kterou chceme seřadit.

4 Analýza systému pro testování

4.1 Specifikace požadavků

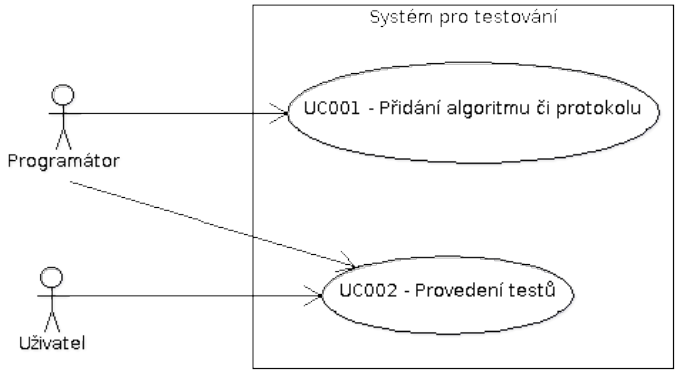
Výsledná aplikace pro testování by měla umožňovat porovnávání rychlostí různých implementovaných algoritmů pro různá vstupní data, za použití různého počtu paralelních vláken a různých komunikačních protokolů (v distribuovaném prostředí).

Aplikace by dále měla nabízet intuitivní ovládání pro použití i neznalým uživatelem, ideálně za pomoci grafického uživatelského rozhraní. Celý program by měl být napsán dostatečně modulárně tak, aby umožňoval programátorovi snadno implementovat další komunikační protokoly či algoritmy.

Celá aplikace by měla být rozdělena do logických celků (balíčků) dle účelu a použití a za účelem dosažení určité úrovně modularity využívat rozhraní a abstraktních tříd.

4.2 Případy užití

Aplikace najde využití především v případě potřeby porovnání rychlosti implementovaných algoritmů v paralelním a distribuovaném prostředí s využitím implementovaných protokolů, dále nabízí programátorovi možnost implementace dalších algoritmů či komunikačních protokolů (viz obr. 4.1).



Obrázek 4.1: UML diagram případů užití

- *UC001* – Programátorovi systém nabízí možnost přidání dalších algoritmů či komunikačních protokolů.
- *UC002* – Programátorovi i běžnému uživateli systém umožňuje porovnat rychlost implementovaných algoritmů, a to v paralelním prostředí s využitím proměnného počtu vláken, či v distribuovaném prostředí s využitím různých nízkoúrovňových i vysokoúrovňových komunikačních protokolů. Zmíněná srovnání rychlosti mohou být použita pro zjištění „výhodnosti“ použití konkrétních algoritmů či protokolů v daných prostředích pro různě velká vstupní data.

4.3 Vybrané technologie a postupy

Po nastudování jednotlivých vlastností, předností a záporů protokolů zmíněných v kapitole 2.2 jsem s ohledem na jejich odlišnosti zvolil konkrétní tři pro následnou implementaci:

4.3.1 TCP

TCP protokol jsem zvolil především pro jeho velmi široké použití a rozšíření v nejrůznějších oblastech a jakožto zástupce „nižších“ protokolů. U tohoto protokolu očekávám především nižší režii a lehce obtížnější implementaci než při použití některého vysokoúrovňového řešení (viz 2.2.1).

4.3.2 (TCP) NIO

New I/O API s využitím zmíněného základního protokolu TCP jsem zvolil především pro vhodnost jeho porovnání se základním TCP protokolem a ověřením, zda splňuje vlastnosti pro které byl navržen. Při implementaci tohoto protokolu tedy očekávám obecně vyšší rychlosti přenosu dat než u základního TCP za cenu obtížnější implementace (viz 2.2.3).

4.3.3 RMI

RMI API, konkrétně jeho variantu s protokolem JRMP jsem zvolil především jako zástupce vyššího prostředku pro distribuci dat a jednak pro jeho „vyhrazenost“ pro jazyk Java (obdobně jako NIO). Protože se jedná o obecně vysokoúrovňový prostředek, očekávám při použití tohoto přístupu především vyšší režii při přenosu dat, ale na druhou stranu mnohem jednodušší implementaci než u předchozích řešení (viz 2.2.5).

5 Implementace systému

5.1 Popis balíků a tříd

Celá aplikace pro testování zmíněných algoritmů za použití daných protokolů je napsána pro verzi Javy 1.7.0 a rozdělena do několika balíků (viz obr. 5.1).

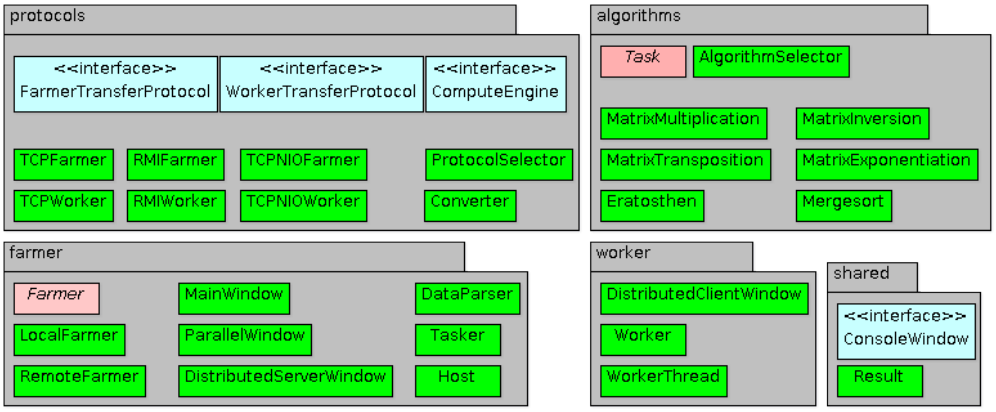
Balík `algorithms` obsahuje jednak především rozhraní (interface) `Task`, který následně implementují všechny použité algoritmy a dále samotné implementace algoritmů. Do balíku je možné v budoucnu snadno přidat další vlastní algoritmy implementací zmíněného rozhraní `Task` a následným zápisem do třídy `AlgorithmSelector`.

Balík `farmer` obsahuje vše, co je specifické pro paralelní výpočet (na jednom zařízení) a dále pro řídící proces distribuovaného výpočtu.

Balík `protocols` obsahuje (jak již napovídá název) rozhraní a konkrétní implementace jednotlivých přenosových protokolů (tj. TCP, NIO, RMI). Jednotlivé protokoly jsou díky implementaci jednotlivých rozhraní „navenek“ značně uniformní, čili je lze používat až na drobné odchylky naprosto stejným způsobem, voláním stejných metod.

Balík `shared` obsahuje několik drobnějších tříd, které jsou společné všem hlavním částem aplikace – tedy paralelnímu prostředí a oběma stranám distribuovaného prostředí. Za zmínku stojí např. třída `Result`, která představuje obecný výsledek úlohy.

Balík `worker` je protipólem balíku `farmer`, obsahuje tedy veškeré třídy a rozhraní specifické pro výpočetní proces distribuovaného prostředí.



Obrázek 5.1: UML diagram balíků (bez znázornění vazeb)

5.2 Paralelní prostředí

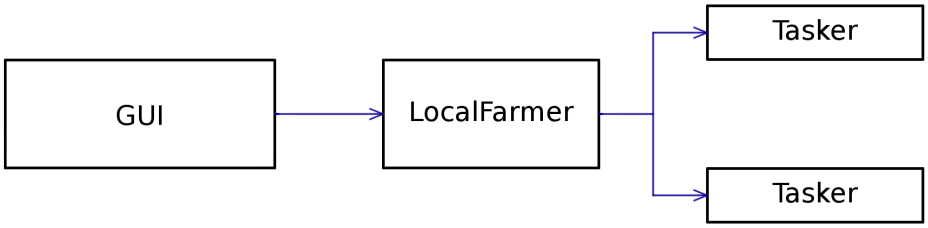
Postup celého programu je při použití paralelního a distribuovaného prostředí v mnohém obdobný. Základ tvoří třídy `Task` a `Result`, jejichž instance reprezentují jednotlivé výpočetní úlohy a výsledky. Objekty těchto typů jsou také jediné, které se při použití distribuovaného výpočtu přenášejí po síti mezi jednotlivými procesy.

Abstraktní třída `AFarmer` slouží jako základ pro třídy `LocalFarmer` a `RemoteFarmer`, které (jak již název napovídá) slouží pro řízení výpočtu v paralelním, resp. distribuovaném prostředí. Tyto třídy dále využívají několika podpůrných tříd pro různé operace, jmenovitě např. třídu `DataParser` pro načítání a zpracování vstupních dat pro jednotlivé úlohy.

Paralelní výpočet probíhá především za pomoci tříd z balíku `farmer` (především `LocalFarmer` a `Tasker`), které při paralelním výpočtu zastávají roli řídící i výpočetní. Při spuštění výpočtu z grafického rozhraní se o samotné rozdělení úloh mezi jednotlivá vlákna stará třída `LocalFarmer`, která udržuje a přiděluje úlohy instancím třídy `Tasker` (viz obr. 5.2). Ta slouží jako výkonná část paralelního výpočtu – dědí od nativní třídy `Thread` a po spuštění spustí výpočet samotné úlohy, načež vrátí výsledek zpátky třídě `LocalFarmer`. Pokud je po navrácení výsledku dostupná ještě nějaká úloha, je tato

ihned přiřazena další volné instanci třídy `Tasker`.

Třída `LocalFarmer` se mimo jiné stará také o výpočet celkového času výpočtu a ukládání výsledků úloh do souboru (opět pomocí třídy `DataParser`).



Obrázek 5.2: Znázornění práce paralelního výpočtu při použití 2 výpočetních vláken

5.3 Distribuované prostředí

Výpočet v distribuovaném prostředí je značně složitější, nebot' se zde využívá prakticky všech prostředků používaných při paralelním výpočtu a navíc prostředků pro komunikaci mezi vzdálenými procesy.

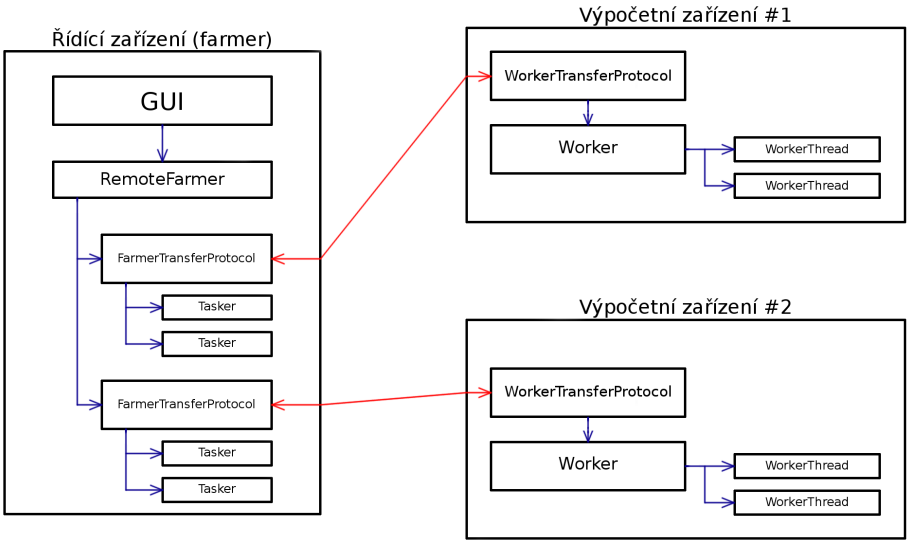
Základ tvoří třída `RemoteFarmer`, což je obdoba `LocalFarmer` zmíněného v 5.2. Tato třída následně vytváří pro každý vzdálený proces instanci typu `FarmerTransferProtocol`, což je rozhraní, které splňují všechny implementované protokoly a které se používá pro samotnou komunikaci. Veškeré ostatní třídy komunikují se vzdálenými zařízeními výhradně pomocí instancí tříd implementující rozhraní `FarmerTransferProtocol` a `WorkerTransferProtocol`.

Zmíněné třídy dále na základě nastaveného počtu výpočetních vláken vytvářejí instance třídy `Tasker`, které v tomto případě slouží pro samotnou „obsluhu“ jednotlivých úloh – v praxi tedy dostává každá instance třídy `Tasker` přidělen jeden konkrétní úkol (podobně jako v paralelním prostředí) a stará se následně o její přenos přes síť, přijetí výsledku a předání zpět řídícímu procesu (`RemoteFarmer`).

Na výpočetních zařízeních se o komunikaci starají objekty typu `WorkerTransferProtocol` – obdoba `FarmerTransferProtocol`. O samotné řízení výpočtu na jednotlivých zařízeních se stará třída `Worker`, která jednotlivé přijaté úlohy předává instancím třídy `WorkerThread`. Ty provedou výpočet a následně předají výsledek zpět třídě `Worker`, která je následně předá opět instanci `WorkerTransferProtocol` k odeslání zpět hlavnímu řídícímu procesu.

Výpočetní zařízení představují v tomto systému servery, které na počátku čekají na zvoleném protokolu, adrese a portu na požadavky, zatímco výpočetní zařízení funguje jako klient, který se připojuje k těmto serverům a následně jim přiděluje jednotlivé úkoly. Výpočetní procesy jsou vytvořeny tak, že ihned po přijetí úlohy vytvoří novou instanci `WorkerThread` a spustí výpočet – hlavní řídící proces (farmer) tedy pošle „najednou“ každému výpočetnímu procesu tolik úloh, kolik je nastaveno výpočetních vláken.

Řídící proces přiděluje vždy po přijetí výsledku další úlohu do doby, než odešle předem přednastavený počet úloh (pokud byl nastaven) nebo všechny úlohy v zadaném vstupním souboru. Výpočetní procesy se po zpracování všech úloh nacházejí v prvotním stavu čekání na nové spojení.



Obrázek 5.3: Znázornění práce distribuovaného výpočtu s použitím 2 výpočetních zařízení a 2 výpočetních vláken na každém z nich

6 Implementace algoritmů

Kromě samotného systému bylo potřeba implementovat i několik testovacích algoritmů.

Protože jsem se na konec rozhodl pro použití několika existujících implementací třetích stran, nemohl jsem zcela naplnit původní myšlenku, kdy by u každého algoritmu docházelo k rozdělování jednotlivých úloh (např. jedné matice či jedné posloupnosti čísel) do menších „podúloh“ (jednotlivé řádky a sloupce matice či části posloupnosti), tyto byly řešeny nezávisle a posléze opět složeny do konečného výsledku. Z důvodu udržení určité uniformity jednotlivých algoritmů jsem se tedy rozhodl algoritmy implementovat (a následné testy provádět) tak, že místo distribuce jedné velké úlohy dělené do menších částí a následného složení do jednoho výsledku se bude distribuovat více menších úloh, jejichž výsledky budou uloženy nezávisle – nedojde tedy k prvnímu kroku rozdělení úlohy na menší úlohy a poslednímu kroku opětovného složení jednotlivých výsledků. V praxi se tedy například místo distribuce částí jedné matice o rozměrech 10000×10000 bude distribuovat několik samostatných matic o rozměru 1000×1000.

Tento fakt částečně mění vzájemné odlišnosti v komunikační náročnosti jednotlivých algoritmů při distribuci, nikterak však nebrání možnosti stále tyto algoritmy z daného hlediska porovnávat, neboť i náročnost distribuce celých úloh se u jednotlivých algoritmů značně liší – například distribuce dat pro výpočet vynásobení dvou matic je obecně značně náročnější než distribuce dat pro algoritmus Eratosthenova síta (viz kapitola 3).

6.1 Algoritmy

Z algoritmů pro testování uvedených v kapitole 3 jsem se rozhodl pro implementaci následujících šesti.

- *Inverze matice* – pro implementaci tohoto algoritmu využívám existující implementace pro inverzi matice v knihovně `Commons Math` z projektu `Apache Commons`. Jednotlivé matice jsou reprezentovány objekty typu

`Array2DRowRealMatrix` a konkrétní užitá implementace vypočítává inverzní matici za pomoci LU rozkladu[31].

- *Násobení matic* – pro tento algoritmus opět využívám existující implementace ve zmíněné knihovně `Commons Math`, konkrétně metody `multiply()` rozhraní `RealMatrix`, kterou implementuje dříve zmíněná třída `Array2DRowRealMatrix`[31].

- *Umocňování matice* – implementace tohoto algoritmu je téměř identická algoritmu násobení matic, je tedy využito stejných prostředků jako u předchozího bodu.

- *Transpozice matice* – do čtveřice využiji zmíněné matematické knihovny, konkrétně metody `transpose()` již zmíněného rozhraní `RealMatrix`[31].

- *Eratosthenovo síto* – Eratosthenovo síto implementuji jednoduše průchodem pole o velikosti všech čísel v daném rozmezí a aplikací pravidel zmíněných v části 3.7.1. Algoritmus by mohl být částečně urychlen pokud by procházel čísla pouze do hodnoty odmocniny ze zadané horní meze (protože po aplikaci zmíněných pravidel už se ve zbylé posloupnosti žádné další neprvočíslo nemůže nacházet), nicméně pouze pro účely porovnávání rychlosti nemá takové urychlení žádný význam.

- *Mergesort* – pro implementaci tohoto algoritmu využívám vnitřní metody jazyka Java, konkrétně metody `sort()` třídy `Collections`. Tato metoda využívá (ve verzi Java 1.7) stabilní a adaptivní variantu klasického mergesort algoritmu, která vyžaduje značně méně než (n $\times$ log `n`) porovnání v případě částečně seřazené posloupnosti a v případě náhodně seřazené posloupnosti poskytuje výkon ekvivalentní základní implementaci[7].

6.2 Vstupní data

Formát vstupních dat je pro všechny algoritmy do určité míry jednotný. Vstupní data jsou uložena v textových souborech, přičemž v jednom souboru se (typicky) nachází data pro několik úloh stejného algoritmu (např. 20 párů matic pro algoritmus násobení matic). Jednotlivé úlohy jsou v souboru ukončeny (odděleny) samostatným řádkem obsahujícím sekvenci tří hvězdiček ($***$). Formát dat jednotlivých úloh je pak individuální pro každý algoritmus – třída `DataParser` se stará pouze o oddělení dat pro jednotlivé

úlohy a ty pak předává (ve formě `String` objektu) k dalšímu zpracování třídě konkrétního algoritmu. Třída `DataParser` je při zpracování vstupního souboru schopna detekovat některé chyby či nesrovnalosti ve vstupním souboru, např. pokud vstupní soubor nekončí již dříve zmíněnou sekvencí $(***)$ nebo obsahuje méně úloh, než je požadováno uživatelem při spuštění výpočtu.

7 Testy

Veškeré testy jsem prováděl na vstupních datech uměle generovaných pomocí generátoru náhodných čísel, který je součástí Javy (s výjimkou vstupních dat pro Eratosthenovo síto, u kterého by to nedávalo smysl). Pro každou úlohu jsem vygeneroval dvě sady dat, z nichž jedna reprezentuje „malá“ vstupní data (tj. data, pro která zpracování jedné úlohy na dnešním průměrném multimediálním počítači zabere zanedbatelný čas) a druhá „velká“ vstupní data (tj. data, pro která zpracování jedné úlohy na zmíněném počítači zabere čas minimálně v řádu stovek milisekund až sekund). Řádově rozdílné velikosti těchto vstupních dat umožnily porovnat režii jednotlivých prostředků a protokolů v obou prostředích. Každá sada použitých vstupních dat obsahovala celkem 20 samostatných úloh. Každá sada testů byla provedena několikrát po sobě a výsledky byly následně zprůměrovány, aby se snížil vliv případných nepřesností.

7.1 Prostředí pro testování

Aby byly testy provedené v paralelním a především v distribuovaném prostředí porovnatelné, bylo nutné použít pro testování sadu počítačů se stejnými parametry. Pro tento účel jsem používal celkem čtyři počítače s následujícími parametry:

- Procesor Intel Core i7 950, základní takt 3.06 GHz, 4 jádra s technologií HyperThreading
- 12 GB paměti RAM,
- připojení do ethernetové sítě rychlostí 1 Gb/s,
- operační systém Windows 7 Professional, 64-bitová verze.

Aby se minimalizoval vliv čtení dat z pevného disku, byla před spuštěním každého testu kompletní vstupní data nejprve načtena do paměti a až následně spuštěn test. U výsledků testů jsem sledoval především dva důležité parametry – celkový čas od spuštění do přijetí posledního výsledku a množství přenesených dat.

7.2 Testy s pevným počtem výpočetních vláken a počítačů

Jako první jsem provedl sadu testů v paralelním i distribuovaném prostředí s následujícími parametry:

- Paralelní prostředí – 1 řídící vlákno + 3 výpočetní vlákna,
- Distribuované prostředí – 1 řídící počítač + 3 výpočetní počítače, 1 výpočetní vlákno na každém výpočetním počítači.

Tabulka 7.1 ukazuje výsledné časy výpočtů 20 úloh pro dané algoritmy v paralelním prostředí a průměrné časy výpočtů jedné úlohy, vypočtené jako součet časů výpočtů každé úlohy vydělené počtem úloh a počtem vláken (nejedná se tedy o čas vypočtený z uvedeného celkového času, ale časů měřených nezávisle samotnými výpočetními vlákny).

Algoritmus (vždy 20 úloh)	Celkový čas	Prům. čas / úloha
Násobení matic 1000 × 1000	15,729 s	0,586 s
Násobení matic 100 × 100	0,218 s	0,006 s
Inverze matic 1000 × 1000	78,617 s	3,042 s
Inverze matic 100 × 100	0,577 s	0,023 s
Transpozice matic 1000 × 1000	7,738 s	0,2075 s
Transpozice matic 100 × 100	0,203 s	0,005 s
Umocňování matic 1000 × 1000 na 50	91,854 s	4,216 s
Umocňování matic 100 × 100 na 2	0,234 s	0,005 s
Mergesort, 10 000 000 prvků	6,240 s	0,211 s
Mergesort, 100 prvků	0,094 s	0 s
Eratosthenovo síto, limit 10 000 000	3,276 s	0,106 s
Eratosthenovo síto, limit 100	0,078 s	0 s

Tabulka 7.1: Výsledky v paralelním prostředí, 3 výpočetní vlákna

Z tabulky 7.1 jsou na první pohled evidentní nesrovnalosti u algoritmů mergesort a Eratosthenova síta, kde průměrné časy výpočtu jedné úlohy pro malá

data vycházejí nulové. To přisuzuji nepřesnosti použité metody `currentTimeMillis()`, která vrací čas v milisekundách, avšak její rozlišovací schopnost je závislá na operačním systému[7]. V těchto případech byly tedy jednotlivé úlohy dokončeny za čas nižší, než je rozlišovací schopnost na použitém systému a naměřený čas každé úlohy byl tak nulový.

Tabulka 7.2 pak ukazuje výsledné časy výpočtů 20 úloh pro dané algoritmy za použití komunikačních protokolů uvedených v kapitole 4.3. Pro úsporu místa jsou vynechány průměrné časy na úlohu. Grafická reprezentace těchto výsledků viz obr. C.1 a C.2.

Alg. (vždy 20 úloh)	TCP	RMI	NIO
Násobení matic 1000 × 1000	35,911 s	29,750 s	26,115 s
Násobení matic 100 × 100	0,686 s	0,359 s	0,483 s
Inverze matic 1000 × 1000	60,294 s	59,297 s	56,803 s
Inverze matic 100 × 100	0,780 s	0,390 s	0,686 s
Transpozice matic 1000 × 1000	22,403 s	18,127 s	15,054 s
Transpozice matic 100 × 100	0,484 s	0,250 s	0,406 s
Umocňování matic 1000 × 1000 na 50	98,952 s	92,087 s	89,655 s
Umocňování matic 100 × 100 na 2	0,499 s	0,265 s	0,452 s
Mergesort, 10 000 000 prvků	47,097 s	21,279 s	20,295 s
Mergesort, 100 prvků	3,104 s	0,063 s	3,136 s
Eratosthenovo síto, limit 10 000 000	11,590 s	9,563 s	11,763 s
Eratosthenovo síto, limit 100	3,058 s	0,063 s	3,120 s

Tabulka 7.2: Výsledky v distribuovaném prostředí, 3 výpočetní počítače

Tabulka 7.3 dále ukazuje velikosti přenesených vstupních a výstupních dat pro jednotlivé algoritmy (jedná se pouze o data pro samotný výpočet, tedy bez započtení jakékoli režie přenosových protokolů – pro všechny protokoly jsou tedy tato data stejná). Grafická reprezentace těchto výsledků viz obr. C.3 a C.4.

Alg. (vždy 20 úloh)	Data vstupů	Data výsledků[2]
Násobení matic 1000 × 1000	1423,35 MB	736,09 MB
Násobení matic 100 × 100	14,24 MB	7,95 MB
Inverze matic 1000 × 1000	711,98 MB	792,31 MB
Inverze matic 100 × 100	7,12 MB	7,93 MB
Transpozice matic 1000 × 1000	711,68 MB	604,93 MB
Transpozice matic 100 × 100	7,12 MB	6,76 MB
Umocňování matic 1000 × 1000 na 50	711,66 MB	871,22 MB
Umocňování matic 100 × 100 na 2	7,12 MB	7,75 MB
Mergesort, 10 000 000 prvků	418,96 MB	408,49 MB
Mergesort, 100 prvků	0,04 MB	0,04 MB
Eratosthenovo síto, limit 10 000 000	0,00 MB[1]	184,44 MB
Eratosthenovo síto, limit 100	0,00 MB[1]	0,00 MB[1]

Tabulka 7.3: Celková velikost dat algoritmů přenesených ve směru k výpočetním počítačům a zpět

Z naměřených hodnot uvedených v tabulkách lze získat několik zajímavých poznatků:

- Nepotvrdil se původní předpoklad, že TCP protokol bude obecně rychlejší než protokol RMI – naopak protokol RMI se oproti TCP ukázal jako rychlejší pro všechny výše zmíněné případy.
- Potvrdil se předpoklad, že protokol NIO (použitý nad protokolem TCP) je obecně rychlejší či minimálně srovnatelný se základním protokolem TCP. Pro velmi malá data jsou protokoly rychlostně téměř ekvivalentní, optimalizace protokolu NIO se však znatelně projevuje s rostoucí velikostí přenášených dat.
- Dle předpokladů se protokol NIO ukázal jako rychlejší oproti protokolu RMI při přenosu velkého množství dat, avšak pro data zanedbatelné velikosti se jako znatelně rychlejší ukázal právě protokol RMI.

[1] Méně než 3 kB.

[2] Určité odlišnosti ve velikostech vstupních a výstupních dat u algoritmů, kde by se jejich velikosti měly rovnat (např. transpozice matice) jsou způsobeny mírně odlišným formátem uložení vstupů a výsledků ve formě `String` objektu.

- Zajímavostí je dále srovnání s paralelním prostředím – ačkoli u většiny algoritmů se výpočet v paralelním prostředí ukázal jako rychlejší, u některých algoritmů se naopak ukázala vhodnost použití distribuovaného prostředí. Například v případě inverze velkých matic se distribuce úloh mezi 3 počítače ukázala jako mnohem vhodnější pro všechny tři použité protokoly, u algoritmu umocňování velkých matic byl pak o trochu rychlejší výpočet při použití protokolu NIO. Tento fakt přisuzuji náročnosti na použití operační paměti – u algoritmů pracujících s velkými daty a intenzivně využívajících operační paměť překoná použití distribuovaného prostředí (kde každé zařízení provádí výpočet za pomoci své vlastní paměti) prostředí paralelní (kde jednotlivá jádra pracují se společnou pamětí).

7.3 Testy s proměnným počtem výpočetních vláken a počítačů

Jako druhou jsem se rozhodl provést sadu testů s použitím pouze dvou vybraných algoritmů a jedné sady vstupních dat pro každý z nich, avšak s proměnným počtem vláken (v paralelním prostředí) a použitých počítačů (v distribuovaném prostředí).

Pro tyto testy jsem se snažil zvolit algoritmy co možná nejvíce odlišné co se týče náročnosti na komunikaci. Na konec jsem zvolil algoritmus násobení matic jako zástupce komunikačně náročného algoritmu a Eratosthenovo síto jako zástupce komunikačně nenáročného algoritmu.[1]

Tabulka 7.4 ukazuje výsledky při použití algoritmu násobení matic rozměru 1000×1000 při použití 1-4 vláken v paralelním prostředí a 1-3 počítačů v distribuovaném prostředí. Tabulka 7.5 pak zobrazuje totéž pro algoritmus Eratosthenova síta s horní mezí 10 000 000. Grafická reprezentace těchto výsledků viz obr. C.5 a C.6.

[1] Ačkoli výsledky tohoto algoritmu mohou být značně velké, vstupní data jsou zanedbatelné velikosti.

Násobení matic 1000 × 1000, 20 úloh				
Počet vláken či počítačů	Paralelní prost.	TCP	RMI	NIO
1	35,398 s	90,154 s	68,813 s	108,365 s
2	18,127 s	53,058 s	45,444 s	39,593 s
3	15,429 s	36,176 s	31,918 s	29,655 s
4	11,778 s	-	-	-

Tabulka 7.4: Rychlost výpočtu 20 úloh násobení matic 1000 × 1000 v paralelním a distribuovaném prostředí při proměnném počtu vláken a počítačů

V tabulce 7.4 je zajímavý především výsledek při použití 1 počítače a protokolu NIO – zatímco při použití více počítačů se tento protokol ukázal jako nejrychlejší, při použití pouze jednoho výpočetního počítače se naopak ukázal jako nejhorší. Mohlo se však také jednat o určitou anomálii, což by mohlo být ověřeno zopakováním celé sady testu vícekrát – v ostatních případech totiž výsledky víceméně korespondují s výsledky v tabulce 7.2. Celkové časy výpočtů v distribuovaném prostředí klesají s rostoucím počtem počítačů zhruba stejnou měrou jako v paralelním prostředí při zvyšování počtu vláken, při použití takto malého počtu vláken či počítačů je pro tento konkrétní algoritmus však vhodné použití paralelního prostředí.

Eratosthenovo síto, limit 10 000 000, 20 úloh				
Počet vláken či počítačů	Paralelní prost.	TCP	RMI	NIO
1	6,631 s	29,984 s	21,949 s	11,623 s
2	4,041 s	18,190 s	17,534 s	15,711 s
3	3,338 s	11,481 s	11,062 s	10,157 s
4	3,120 s	-	-	-

Tabulka 7.5: Rychlost výpočtu 20 úloh Eratosthenova síta s horní mezí 10 000 000 v paralelním prostředí při proměnném počtu vláken

Z výsledků v tabulce 7.5 jsou tentokrát nejzajímavější dva údaje:

- Výsledek pro 1 počítač s použitím protokolu NIO vyšel pro tento algoritmus obráceně, než u algoritmu násobení matic – v tomto případě se zdál až dvakrát rychlejší než protokol RMI. Výkonnost protokolu NIO se tedy zdá na základě těchto výsledků velmi proměnlivá při použití

pouze jednoho výpočetního zařízení, při zvyšování počtu zařízení pak výsledky více kopírují zbylé dva protokoly.

- Při použití 3 počítačů se protokol NIO zdál pro tento konkrétní algoritmus o sekundu rychlejší než protokol RMI, zatímco v předchozím testu (viz tabulka 7.2) se naopak téměř o dvě sekundy ukázal rychlejší protokol RMI. Tyto vzájemné odchylky přisuzuji faktu, že u zvoleného algoritmu se přenáší velice malá data (alespoň směrem k počítačům), což může vést k nižší přesnosti měření (viz kapitola 7.2).

8 Závěr

V rámci této bakalářské práce byly prozkoumány prostředky jazyka Java pro paralelní a distribuované prostředí a následně byla s pomocí několika vybraných prostředků vytvořena aplikace, která umožňuje jednoduchým způsobem porovnávat výkonnost různých algoritmů a komunikačních protokolů ve zmíněných prostředích. V rámci této aplikace bylo za účelem testování implementováno použití tří různých komunikačních protokolů a také šesti algoritmů, především s využitím knihovny `Apache Commons Math`. Algoritmy byly vybrány s ohledem na odlišnou náročnost na komunikaci a složitost výpočtu.

Aplikaci se podařilo realizovat dle původního záměru a celý systém byl navržen tak, aby ho bylo možné snadno rozšířit o další algoritmy či komunikační protokoly.

V rámci provedených testů se ukázala vhodnost použití jednotlivých prostředí a protokolů v závislosti na použitém algoritmu a množství zpracovávaných dat. Protokol NIO se obecně ukázal jako nejvhodnější v případě přenosu velkých množství dat, zatímco protokol RMI se zdá vhodnější při přenosu malých až zanedbatelných množství dat. Přenos pomocí protokolu NIO použitém nad protokolem TCP se také ukázal jako obecně rychlejší, než pomocí samostatného protokolu TCP. Při použití algoritmů intenzivně využívajících operační paměť (např. zvolená implementace inverze matic) se navíc ukázala výhoda použití distribuovaného prostředí před použitím paralelního prostředí s více vlákny.

V práci by bylo možné dále pokračovat například rozšířením systému o další protokoly a algoritmy či implementací měření dalších statistik. Dále by bylo možné rozšířit uživatelské rozhraní za účelem lepší kontroly nad prováděnými výpočty.

Literatura

[1] Doug Lea. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*. Addison Wesley, 1999. ISBN 0201310090.

[2] Jakob Jenkov. java.util.concurrent - Java Concurrency Utilities, . URL `http://tutorials.jenkov.com/java-util-concurrent/index.html`. [cit. 12.6.2014].

[3] Javin Paul. ReentrantLock Example in Java, Difference between synchronized vs ReentrantLock, 2013. URL `http://javarevisited.blogspot.cz/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html`. [cit. 12.6.2014].

[4] Jakob Jenkov. Readwritelock, . URL `http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html`. [cit. 12.6.2014].

[5] Jakob Jenkov. Locks in Java, . URL `http://tutorials.jenkov.com/java-concurrency/locks.html`. [cit. 12.6.2014].

[6] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2008. ISBN 1441418687.

[7] Oracle. Java™ Platform Standard Edition 7, 2014. URL `http://docs.oracle.com/javase/7/docs/api/`. [cit. 12.6.2014].

[8] Tamaki Kurusu. MONITORs in Concurrent Programming, 2000. URL `http://courses.cs.vt.edu/~cs5204/fall00/monitor.html`. [cit. 20.6.2014].

[9] Javin Paul. What is CyclicBarrier Example in Java 5 – Concurrency Tutorial, 2012. URL `http://javarevisited.blogspot.cz/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html`. [cit. 20.6.2014].

[10] Lawrence Abrams. TCP and UDP Ports Explained, 2004. URL http://www.bleepingcomputer.com/tutorials/tcp-and-udp-ports-explained/. [cit. 20.6.2014].

[11] Ashish Myles. Java TCP Sockets and Swing Tutorial, 2006. URL https://www.cise.ufl.edu/~amyles/tutorials/tcpchat/. [cit. 20.6.2014].

[12] Arpit Mandliya. Serialization in Java, 2013. URL http://javapostsforlearning.blogspot.cz/2013/03/serialization-in-java.html. [cit. 20.6.2014].

[13] Jakob Jenkov. Java NIO Overview, . URL http://tutorials.jenkov.com/java-nio/overview.html. [cit. 20.6.2014].

[14] Martin Majer. Síťování v Javě: New I/O, 2006. URL http://www.root.cz/clanky/sitovani-v-jave-new-io/. [cit. 20.6.2014].

[15] Jakob Jenkov. Java NIO Buffer, . URL http://tutorials.jenkov.com/java-nio/buffers.html. [cit. 20.6.2014].

[16] Jakob Jenkov. Java NIO Selector, . URL http://tutorials.jenkov.com/java-nio/selectors.html. [cit. 20.6.2014].

[17] Object Management Group. Corba® basics, 2014. URL http://www.omg.org/gettingstarted/corbafaq.htm. [cit. 16.6.2014].

[18] David Reilly. Java RMI & CORBA, 2006. URL http://www.javacoffeebreak.com/articles/rmi_corba/. [cit. 20.6.2014].

[19] Oracle. Java RMI over IIOP, 2014. URL http://docs.oracle.com/javase/7/docs/technotes/guides/rmi-iiop/. [cit. 20.6.2014].

[20] Oracle. An Overview of RMI Applications, 2012. URL http://docs.oracle.com/javase/tutorial/rmi/overview.html. [cit. 12.6.2014].

[21] Oracle. RMI System Overview, 2010. URL http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-arch5.html. [cit. 20.6.2014].

[22] Nová média, s. r. o. Inverzní matice, 2006. URL http://www.matematika.cz/inverzni-matice. [cit. 16.6.2014].

[23] V. Sahota a R. Bayford. *Jinv: A Parallel Method for Distributed Matrix Inversion.* IEEE, Sch. of Health and Social Sci., Middlesex Univ., London, UK, 2010. ISBN 978-1-4244-8044-9.

[24] Nová média, s. r. o. Matice, 2006. URL http://www.matematika.cz/matice. [cit. 16.6.2014].

[25] Razvan C. Bunescu. Design and Analysis of Algorithms: Lecture 12, 2013. URL http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture12.pdf. [cit. 20.6.2014].

[26] Paul Rademacher. Ray Tracing: Graphics for the Masses, 1997. URL https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html. [cit. 12.6.2014].

[27] Jeffrey J. Holt a John W. Jones. Discovering Number Theory, 2001. URL http://www.math.mtu.edu/mathlab/COURSES/holt/dnt/. [cit. 16.6.2014].

[28] Drahomír Hanák. Eratosthenovo síto, 2012. URL http://www.itnetwork.cz/algoritmus-eratosthenovo-sito. [cit. 16.6.2014].

[29] Petr Berka. Stavový prostor, 2007. URL http://sorry.vse.cz/~berka/docs/4iz430/P01-Stavovyprostor.pdf. [cit. 16.6.2014].

[30] Rashid Bin Muhammad. Merge Sort, 2010. URL http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm. [cit. 16.6.2014].

[31] The Apache Software Foundation. Apache Commons Math 3.3 API, 2014. URL https://commons.apache.org/proper/commons-math/javadocs/api-3.3/. [cit. 12.6.2014].

Seznam obrázků

Seznam tabulek

Seznam zkratek

API Application Programming Interface

CORBA Common Object Request Broker Architecture

GUI Graphical User Interface

I/O Input/Output

IDL Interface Definition Language

IIOP Internet Inter-Orb Protocol

JRMP Java Remote Method Protocol

NIO New I/O

OBR Object Request Broker

RMI Remote Method Invocation

TCP Transmission Control Protocol

TCP/IP Transmission Control Protocol / Internet Protocol

UDP User Datagram Protocol

A Uživatelská příručka

Pro použití programu je nutné mít nainstalované prostředí Java Runtime Environment alespoň ve verzi 1.7.0. Pro překlad ze zdrojových souborů je pak nutné mít nainstalované prostředí Java Development Kit stejné verze. Dále je pak doporučeno mít nainstalován program `Apache Ant`, aby bylo možné využít automatického překladu, vytvoření dokumentace a spustitelného JAR souboru.

A.1 Překlad a spuštění

Přiložený CD-ROM obsahuje veškeré soubory potřebné pro překlad a následné spuštění programu, včetně již vygenerovaného, spustitelného JAR souboru. Všechny tyto soubory jsou uloženy ve složce `program`. Tuto složku si před použitím zkopírujeme z CD-ROM do počítače.

Ve složce se nachází soubor `build.xml` určený pro program `Apache Ant`, který umožňuje automatickou kompilaci, vytvoření dokumentace a vytvoření spustitelného JAR souboru. Pro jeho použití spustíme příkazovou řádku či terminál, přejdeme do složky obsahující zmíněný soubor `build.xml` a zadáme příkaz

```
ant <název cíle>
```

kde `<název cíle>` může být jedno z následujících:

- `clean` – provede úklid, tedy smaže všechny vygenerované soubory a adresáře `bin`, `doc` a `jar`
- `doc` – vytvoří v adresáři `doc` dokumentaci ze zdrojových souborů v adresáři `src`
- `compile` – provede nejprve cíl `clean` a následně přeloží veškeré zdrojové soubory v adresáři `src` do adresáře `bin` a přiloží matematickou knihovnu z adresáře `lib`

- `jar` – provede nejprve cíl `compile` a následně vytvoří v adresáři `jar` spustitelný JAR soubor
- `main` (nebo nic) – provede postupně cíle `compile`, `doc` a `jar`

Po provedení libovolného cíle zahrnující cíl `compile` můžeme pak samotný program spustit z příkazové řádky či terminálu přepnutím se do adresáře `bin` a spuštěním příkazu

```
java farmer.MainWindow
```

V případě spouštění vytvořeného JAR souboru lze pak obdobně tento spustit přepnutím se do adresáře s JAR souborem a spuštěním příkazu

```
java -jar testy.jar
```

A.2 Použití programu

Program je vytvořen za pomoci minimalistického grafického rozhraní a veškeré ovládání je značně intuitivní. Po spuštění programu se zobrazí úvodní okno, umožňující volbu mezi paralelním a distribuovaným prostředím. Při volbě distribuovaného prostředí se navíc zobrazí volby umožňující zvolit mezi řídícím zařízením (farmerem) a výpočetním zařízením (workerem). Volba se jednoduše potvrdí kliknutím na tlačítko OK.

A.2.1 Paralelní výpočet

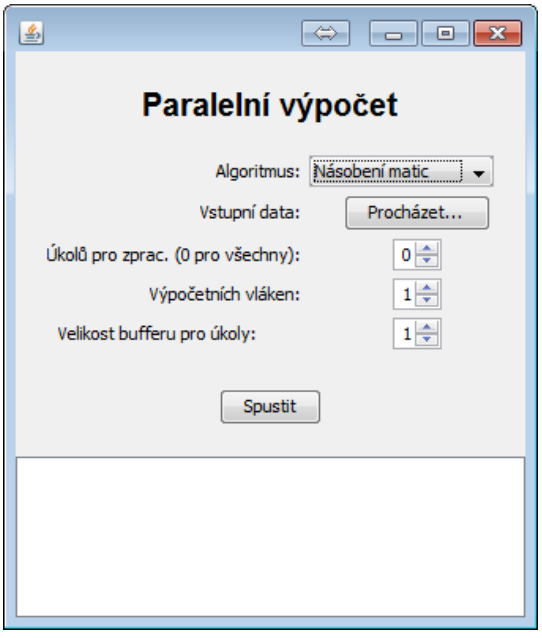
Při volbě paralelního výpočtu se otevře okno, umožňující nastavení jednotlivých parametrů (viz obr. A.2). Z rozbalovacího seznamu s popiskem `Algoritmus` se volí algoritmus, který bude prováděn. Tlačítkem `Procházet...` následně zvolíme soubor vstupních dat pro tento algoritmus. Vzorová vstupní data pro každý algoritmus lze nalézt na přiloženém CD-ROM ve složce `vstupni_data`.

Následují tři pole pro nastavení číselných hodnot:

- `Úkolů pro zprac. (0 pro všechny)` určuje, kolik úkolů ze vstupního souboru bude zpracováváno. Při nastavení hodnoty 0 budou zpracovány všechny úkoly ve vstupním souboru. Při nastavení hodnoty vyšší, než kolik je ve vstupním souboru úkolů, bude před započetím výpočtu zobrazena chyba a výpočet nebude spuštěn.

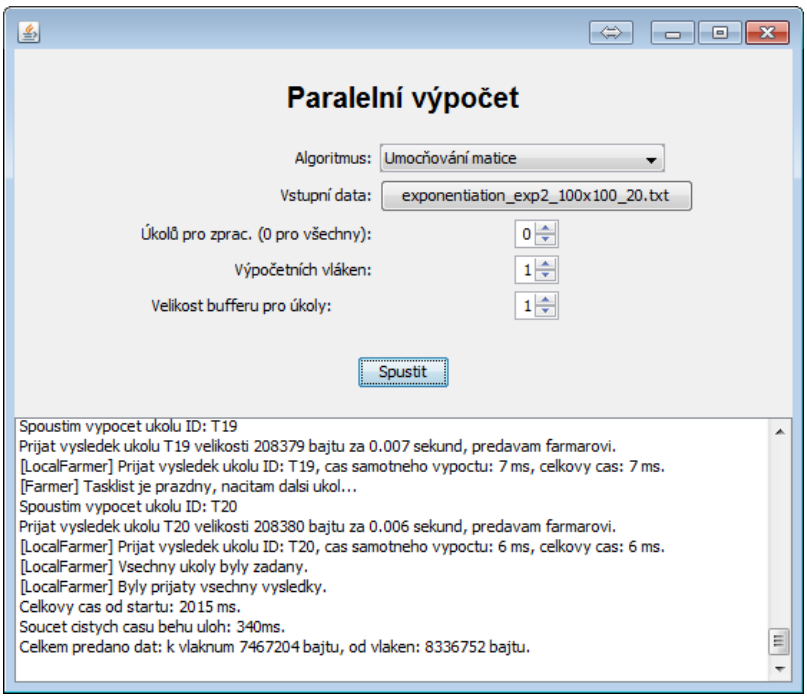
- `Výpočetních vláken` určuje, kolik vláken bude použito pro samotný výpočet. Tato hodnota není shora nikterak omezena, předpokládá se zde určitá disciplinovanost uživatele.

- `Velikost bufferu pro úkoly` určuje, kolik úkolů ze vstupního souboru bude načteno do paměti před započetím samotného výpočtu. Tuto hodnotu je potřeba volit především na základě velikosti vstupního souboru a množství dostupné operační paměti. Při volbě příliš velké hodnoty může dojít k pádu výpočtu z důvodu vzniku chyby `java.lang.OutOfMemoryError`. Tento problém lze do určité míry řešit spuštěním programu s parametrem `-Xmx<heap size limit>`, kde za `<heap size limit>` doplníme množství paměti, kterou může Java pro tento účel využít.



Obrázek A.1: Okno programu pro paralelní výpočet v základním stavu

Kliknutím na tlačítko Spustit se spustí samotný výpočet. Ve spodní části okna se nachází záznam průběhu celého výpočtu, ve kterém jsou průběžně zobrazovány informace o průběhu či chyby. Po skončení výpočtu jsou pak zobrazeny naměřené časy a velikosti vyměněných dat. Výsledky samotných úloh jsou pak ukládány do adresáře, ze kterého byl program spuštěn.



Obrázek A.2: Okno programu pro paralelní výpočet po dokončení výpočtu

A.2.2 Distribuovaný výpočet

Řídící zařízení

Při volbě řídícího procesu distribuovaného výpočtu se zobrazí okno, které se v mnohém podobá oknu pro paralelní výpočet. Navíc je zde rozbalovací seznam pro výběr komunikačního protokolu a pole pro přidávání výpočetních zařízení. Do položek `IP` a `Port` se jednoduše vyplní IP adresa a port, na kterém naslouchá výpočetní zařízení. Kliknutím na tlačítko `Přidat` se pak toto zařízení přidá do seznamu výše. Dvojklikem na zařízení v seznamu ho lze ze seznamu opětovně odebrat. Tímto způsobem lze do seznamu přidat všechna zařízení, která chceme pro výpočet použít. Další postup je již identický s po-

užitím paralelního prostředí (viz A.2.1) – kliknutím na tlačítko Spustit se spustí výpočet a ve spodní části okna je průběžně zobrazován postup a výsledky.

Výpočetní zařízení

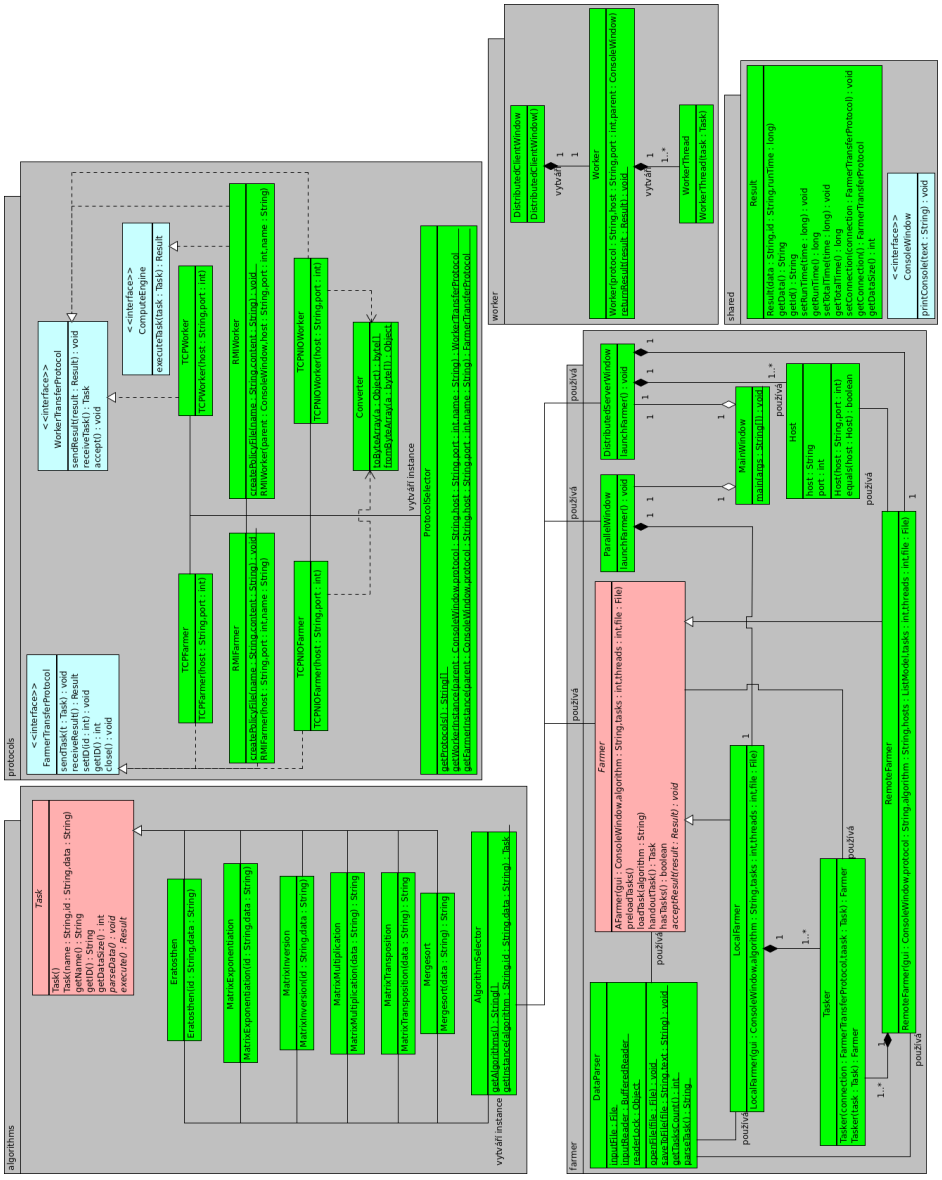
Okno pro konfiguraci výpočetního zařízení je velmi strohé – obsahuje pouze pole pro volbu komunikačního protokolu, adresy pro naslouchání[1] a portu. Toto nastavení musí samozřejmě korespondovat s nastavením na řídícím zařízení. Po kliknutí na tlačítko Spustit se pak počítač pokusí naslouchat na dané adrese a portu s využitím zadaného protokolu a v případě úspěchu je pak připraven přijímat úkoly od řídícího zařízení. Ve spodní části okna jsou opět zobrazovány různé informace o průběhu výpočtu.

A.2.3 Řešení možných problémů

V některých případech (zejména při použití protokolů TCP a NIO) může dojít k situaci, kdy není možné po dokončení jedné sady úloh spustit výpočet sady další – výpočet se spustí, ale nikterak nepostupuje. Tento problém lze vyřešit zavřením a opětovným spuštěním aplikace.

[1] U protokolu RMI výpočet nefunguje korektně při použití univerzální adresy 0.0.0.0 – je tedy vhodné nastavit přímo konkrétní adresu / hostname.

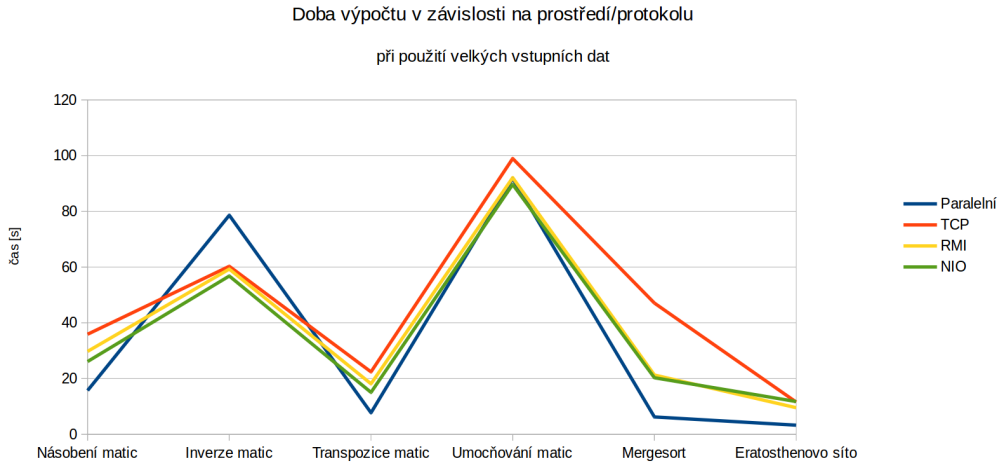
B UML diagram



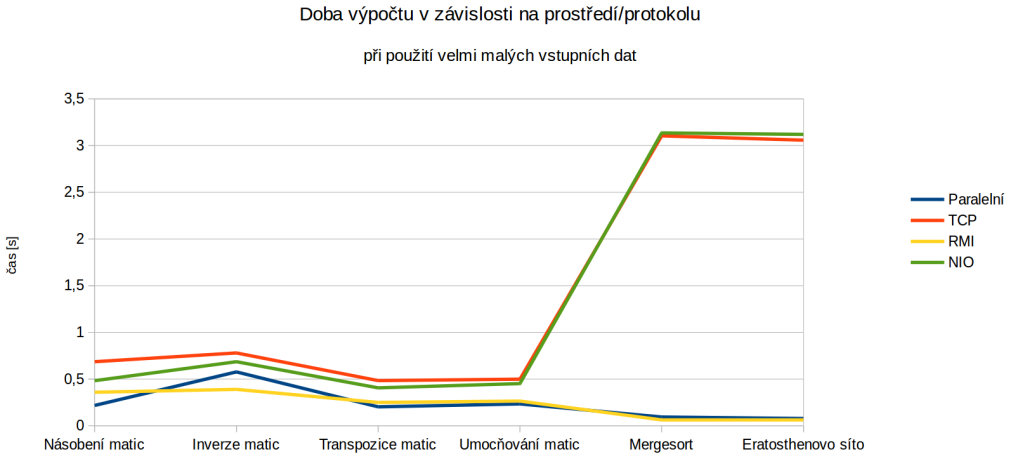
Obrázek B.1: UML diagram tříd
[1]

[1]Digitální verze ve vysokém rozlišení je přiložena na CD-ROM. V diagramu jsou v zájmu přehlednosti vynechány některé méně důležité vazby, zejména týkající se balíků `shared` a `worker`.

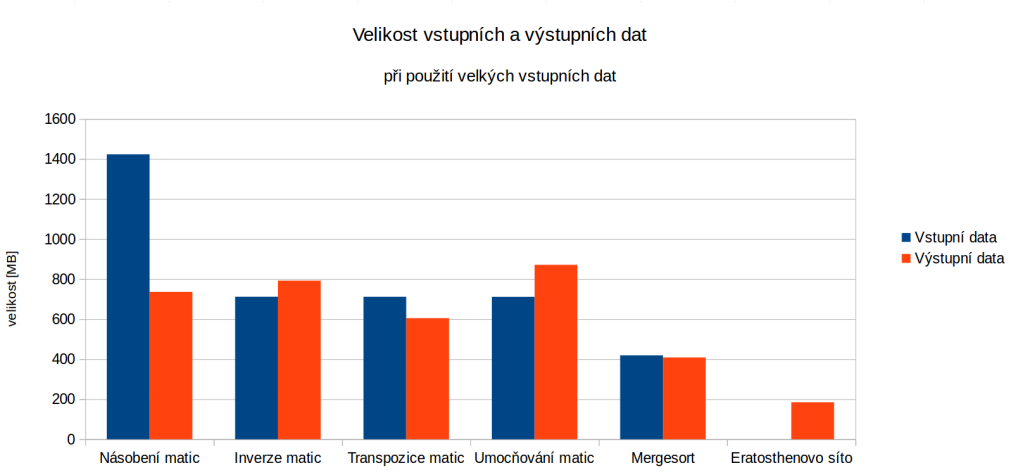
C Grafy výsledků testů



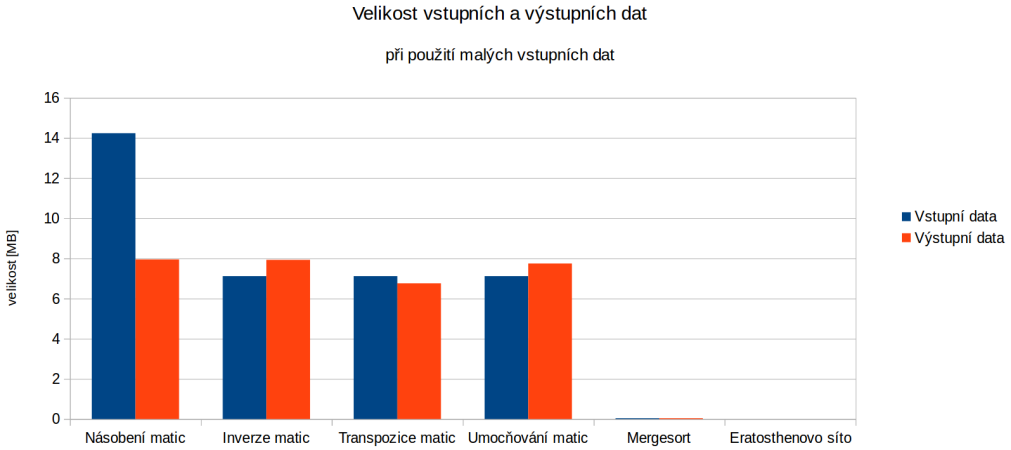
Obrázek C.1: Graf závislosti doby výpočtu na použitém prostředí/protokolu, velká vstupní data



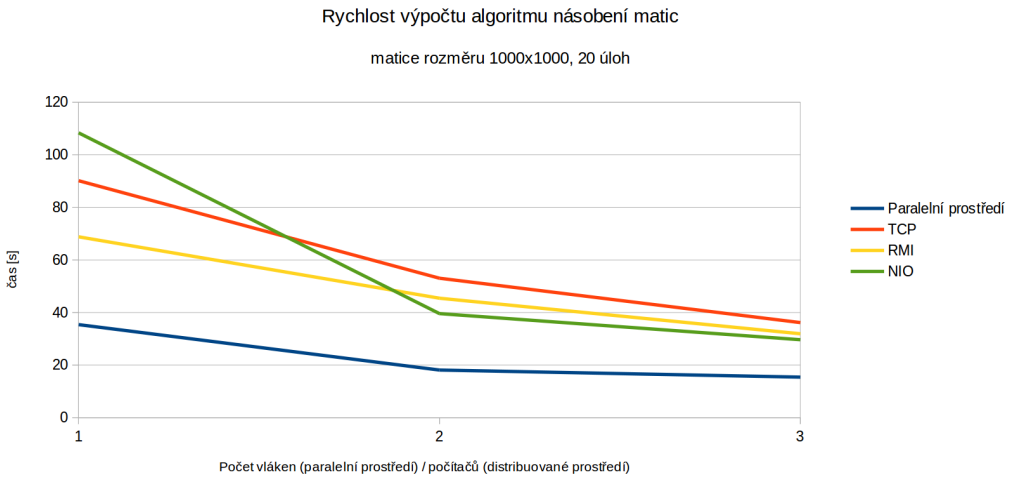
Obrázek C.2: Graf závislosti doby výpočtu na použitém prostředí/protokolu, malá vstupní data



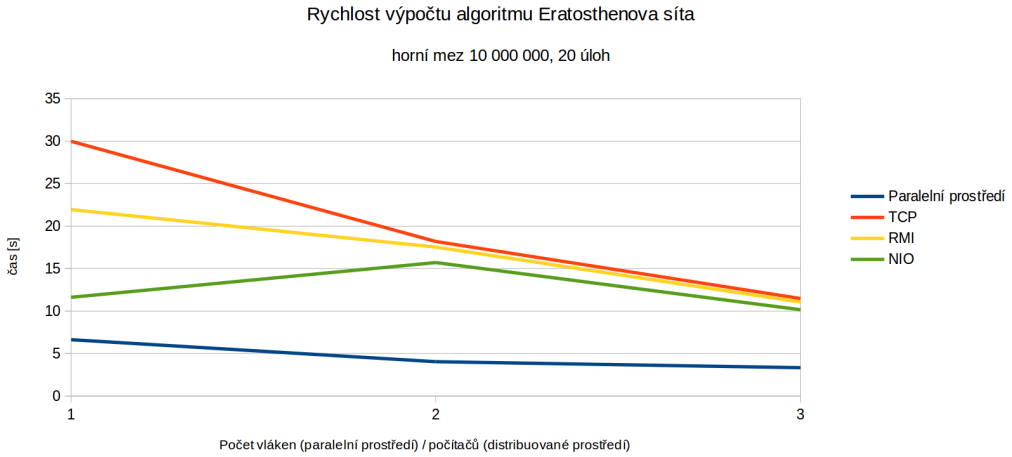
Obrázek C.3: Graf velikosti vstupních a výstupních dat při použití velkých vstupních dat



Obrázek C.4: Graf velikosti vstupních a výstupních dat při použití malých vstupních dat



Obrázek C.5: Graf doby výpočtu algoritmu násobení matic v jednotlivých prostředích při proměnném počtu vláken / počítačů



Obrázek C.6: Graf doby výpočtu algoritmu Eratosthenova síta v jednotlivých prostředích při proměnném počtu vláken / počítačů