UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Porovnání sériové a paralelní Gaussovy eliminační metody v distribuované paměti

Bc. Ondřej Charvát

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2012/2013

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej Charvát**
Osobní číslo:	**I10386**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Porovnání sériové a paralelní Gaussovy eliminační metody v distribuované paměti**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Teoretická část:
V teoretické části práce bude proveden úvod do problematiky paralelních výpočtů, jejich případné optimalizace a vysvětlení základních pojmů. Dále zde bude vysvětlen teoretický postup paralelizace Gaussovy eliminační metody.

Praktická část:
Hlavním cílem práce je implementace vybrané metody paralelizace Gaussovy eliminační metody, její otestování na serveru Hopper ve výzkumném centru NERSC a porovnání získaných dat se sériovou metodou Gaussovy eliminace.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

NERSC homepage [online], 2012 [cit. 3.10.2012]. **Dostupné z WWW: www.nersc.gov**

Top 500 supercomputers [online], 2012 [cit. 3.10.2012]. **Dostupné z WWW: www.top500.org**

Ananth Grama, Anshul Gupta, George Karypis a Vipin Kumar: Introduction to Parallel computing (2nd edition), 2003, ISBN: 978-0-201-64865-2

Daniel Merkle: Parallel Computing (Fall 2011). Dostupné z WWW: http://www.imada.sdu.dk/ daniel/

Vedoucí diplomové práce: **RNDr. Josef Rak**
Katedra matematiky a fyziky

Datum zadání diplomové práce: **31. října 2012**
Termín odevzdání diplomové práce: **17. května 2013**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2012

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 1. 5. 2013 Bc. Ondřej Charvát

Poděkování

Tímto bych rád poděkoval RNDr. Josefu Rakovi za poskytnutí cenných rad a připomínek v průběhu vypracování diplomové práce. Dále bych rád poděkoval profesoru Danielu Merkle a dánské univerzitě University of Southern Denmark za umožnění přístupů na servery NERSC. V neposlední řadě bych rád poděkoval organizaci National Energy Research Scientific Computing Centre za poskytnutí prostředí Hopper pro měření dat pro praktickou část této diplomové práce.

Anotace

Tato diplomová práce se zabývá problematikou paralelního programování a možností jeho uplatnění při řešení soustav lineárních algebraických rovnic.

V praktické části práce je implementována vybraná metoda paralelizace Gaussovy eliminační metody a je provedeno srovnání s metodou sériovou.

Klíčová slova

paralelní programování, Gaussova eliminační metoda, paralelizace

Title

Comparison of Serial and Parallel Gaussian Elimination in a Distributed Memory

Annotation

This thesis is focused on parallel computing and their importance for the solving of system of linear algebraic equations.

The practical part presents an implementation of a chosen method of parallelization of the Gaussian elimination method and its comparison with a serial method of implementation.

Keywords

parallel computing, Gaussian elimination method, parallelization

Obsah

Seznam zkratek ... **8**

Seznam obrázků ... **9**

Seznam tabulek ... **9**

Úvod ... **10**

1 Paralelní programování ... **11**

1.1 Motivace pro zavedení paralelních principů v IT ... 11

1.2 Základní pojmy ... 12

1.2.1 Proces ... 12

1.2.2 Vlákno ... 12

1.3 Paralelizace úlohy ... 13

1.3.1 Vhodná míra abstrakce ... 13

1.3.2 Princip lokality ... 14

1.3.3 Dostatečný výskyt paralelismů ... 14

1.4 Rozdělení paralelních přístupů ... 14

1.4.1 Programovací model se sdílenou pamětí ... 15

1.4.2 Programovací model s distribuovanou pamětí ... 16

1.5 Komunikace a synchronizace procesů ... 17

1.6 Hodnocení paralelních a sériových algoritmů ... 19

1.6.1 Čas běhu vs. čas neefektivity ... 20

1.6.2 Zrychlení ... 20

1.6.3 Efektivita ... 21

1.6.4 Optimální „cena“ ... 21

1.6.5 Vliv granularity a škálovatelnosti ... 22

1.6.6 Vliv hierarchie pamětí ... 23

2 Gaussova eliminace ... **26**

2.1 Základní popis a význam metody ... 26

2.2 Obecný postup eliminace ... 26

2.3 Zpětná substituce ... 27

3 Implementace Gaussovy eliminace ... **28**

3.1 Sériová metoda implementace ... 28

3.1.1 Popis algoritmu ... 28

3.1.2 Stanovení složitosti algoritmu ... 30
3.2 Paralelizace a paralelní implementace ... 30
3.2.1 1-D dělení ... 31
3.2.2 Modifikace 1-D dělení ... 35
3.2.3 Čas paralelního běhu $\boldsymbol{TP}$ ... 35
3.2.4 Knihovna MPI ... 37
3.3 Implementace zpětné substituce ... 38
3.3.1 Sériová implementace zpětné substituce ... 38
3.3.2 Paralelní implementace zpětné substituce ... 39
3.4 Ověření správnosti implementací ... 40
4 Porovnání sériové a paralelní implementace Gaussovy eliminace ... 41
4.1 Popis použitého prostředí v centru NERSC ... 41
4.1.1 Specifikace prostředí Hopper ... 41
4.1.2 Spouštění úloh ... 41
4.2 Metody měření časů ... 44
4.3 Porovnání teoretické a implementované sériové metody ... 45
4.3.1 Definice teoretické sériové metody ... 45
4.3.2 Porovnání teoretického algoritmu s implementovaným ... 45
4.4 Porovnání implementované sériové a implementované paralelní metody ... 48
4.4.1 Celkový čas běhu ... 48
4.4.2 Čas věnovaný pouze výpočtům ... 49
4.4.3 Zrychlení ... 50
4.4.4 Efektivita ... 52
4.4.5 Výpočetní časy jednotlivých procesů ... 54
4.4.6 Náročnost implementace vs. velikost problému n ... 55
4.5 Porovnání teoretické sériové a implementované paralelní metody ... 56
4.5.1 Časy běhu ... 56
4.5.2 Zrychlení ... 57
4.5.3 Efektivita ... 58
Závěr ... 59
Literatura ... 61
Příloha A – Časy běhů jednotlivých procesů pro n = 2000 a p = 4 ... 63
Příloha B – Časy běhů jednotlivých procesů pro n = 2000 a p = 500 ... 64

Příloha C – Časy běhů jednotlivých procesů pro n = 2000 a p = 1000 65

Seznam zkratek

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
DRAM	Dynamic Random Access Memory
FLOP	Floating-point Operation
HD	Z anglického Hard Disk - pevný disk počítače
HW	Hardware
IP	Internet Protocol
IT	Informační technologie
MPI	Message Passing Interface
NERSC	National Energy Research Scientific Computing Centre
SETI	Z anglického Search for Extra Terrestrial Intelligence – hledání mimozemské inteligence
SRAM	Static Random Access Memory
SW	Software

Seznam obrázků

Obrázek 1 – Průměrná výkonnost procesorů od r. 1970 11
Obrázek 2 - Vlákna vs. procesy 13
Obrázek 3 - Programovací model se sdílenou pamětí 15
Obrázek 4 - Programovací model s distribuovanou pamětí 16
Obrázek 5 - Uváznutí procesů 17
Obrázek 6 - Broadcast odeslání zprávy pomocí knihovny MPI v gridové topologii 19
Obrázek 7 - Vliv škálovatelnosti paralelního systému 23
Obrázek 8- Algoritmus sériové implementace Gaussovy eliminace 28
Obrázek 9 - Problém s neaktivní oblastí v průběhu Gaussovy eliminace 31
Obrázek 10 - 1-D rozdělení matice A 32
Obrázek 11 - Distribuce výsledků dělícího kroku 33
Obrázek 12 - Eliminační krok paralelní implementace Gaussovy eliminace 34
Obrázek 13 - Sériová implementace zpětné substituce 38
Obrázek 14 - Zrychlení S versus počet procesů p 51
Obrázek 15 - Výpočetní čas běhu jednotlivých procesů 54

Seznam tabulek

Tabulka 1 - Přehled typů pamětí 24
Tabulka 2 - Specifikace prostředí Hopper 41
Tabulka 3 - Parametry při spouštění úloh na serveru Hopper 43
Tabulka 4 - Porovnání teoretické a implementované sériové metody 46
Tabulka 5 - Celkový čas běhu implementací v sekundách 48
Tabulka 6 - Čas věnovaný pouze výpočtům - % z celkového času běhu 49
Tabulka 7 - Srovnání zrychlení sériové a paralelní implementace 50
Tabulka 8 - Porovnání efektivity sériové a paralelní implementace 52
Tabulka 9 - Porovnání časů běhu teoret. sériové a paralelní implementované metody 57
Tabulka 10 - Porovnání zrychlení teoret. sériové a paralelní implementované metody 57
Tabulka 11 - Porovnání efektivity teoret. sériové a paralelní implementované metody 58

Úvod

Tato diplomová práce se zabývá problematikou paralelního programování. Cílem práce je naprogramování vybrané paralelní implementace Gaussovy eliminační metody a její porovnání s implementací sériovou.

V první kapitole s názvem Paralelní programování je proveden úvod do technologie paralelního programování a jsou zde vysvětleny základní principy a motivace použití paralelizace při řešení výpočetních úloh v informačních technologiích.

Následující kapitola Gaussova eliminace se zabývá vysvětlením základní teorie a matematického postupu Gaussovy eliminační metody a jejího praktického významu.

Třetí kapitola této diplomové práce, která je pojmenována Implementace Gaussovy eliminace, se poté věnuje metodám konkrétní implementace Gaussovy eliminace. Jsou zde především popsány použité algoritmy sériové a paralelní metody a je vypočtena jejich složitost. Dále je zde vysvětlena použitá technika paralelizace a zmíněna nezbytná knihovna potřebná pro tuto techniku.

Čtvrtá kapitola s názvem Porovnání sériové a paralelní implementace Gaussovy eliminace je složena ze třech částí. První je porovnání skutečné sériové implementace se zvolenou teoretickou sériovou metodou. Druhá část se zabývá porovnání skutečně implementované sériové a skutečně implementované paralelní metody z pohledu času běhu, zrychlení, efektivity a času věnovaného pouze výpočtům. Dále je zde ještě zmíněno hledisko „výhodnosti“ a reálnosti použití paralelní implementace vzhledem k velikosti řešeného problému n a hledisko náročnosti implementace. Součástí této kapitoly je také popis experimentu, který se zabýval časy běhů jednotlivých procesů a vlivu na celkovou efektivitu implementace. Třetí a současně poslední částí této kapitoly je porovnání stanovené teoretické sériové metody a skutečně implementované paralelní metody Gaussovy eliminace z pohledu času běhu, zrychlení a částečně efektivity.

Poslední kapitolou je Závěr, kde jsou shrnuty základní poznatky a konkluze získané při vypracování celé práce.

1 Paralelní programování

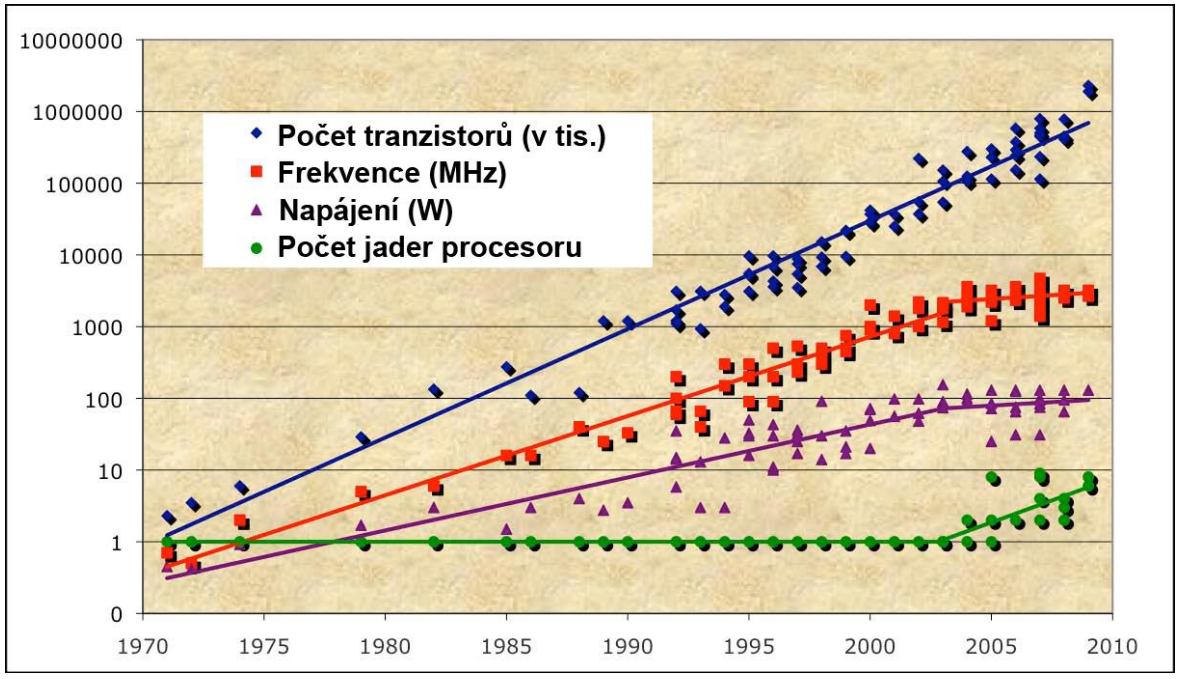
1.1 Motivace pro zavedení paralelních principů v IT

Již v úvodu teoretické části práce je nezbytné vysvětlit základní motivaci pro zavedení paralelního přístupu při řešení úloh v informačních technologiích.

Zvyšující se náročnost a rozsáhlost řešených problémů klade čím dál větší požadavky jak na použitý hardware, tak na použitý software. Některé úkoly se tak mohou za použití běžných sériových metod stát v reálném čase neřešitelné. Příčiny velké náročnosti je možné rozdělit do několika základních kategorií:

- časově náročné operace,
- výpočetně náročné úlohy,
- hranice maximální výkonnosti HW jednoho fyzického stroje,
- zpracování velkých objemů dat,
- potřeba velmi vysoké přesnosti výsledků,
- potřeba získání výsledků v krátkém (reálném) čase,
- hledisko spolehlivosti,
- další zde neuvedené důvody (Ananth Grama, 2003).

Na následujícím obrázku (Obrázek 1) jsou graficky zobrazeny průměrné hodnoty počtu tranzistorů, frekvencí, příkonů napájení a počtu jader průměrných procesorů pro roky 1970 až 2010.



Obrázek 1 – Průměrná výkonnost procesorů od r. 1970 (upraveno dle MERKLE, 2012)

Z tohoto grafu je naprosto patrná tendence kompenzovat rostoucí nároky nových aplikací zvyšováním frekvence procesoru a počtu tranzistorů v jeho jádře. Tento způsob optimalizace bylo možné uplatňovat až do přelomu roku 2003 / 2004, kdy došlo následkem tohoto trendu k dosažení téměř 100 W příkonu procesoru, což je velmi problematická hodnota z pohledu stability a možnosti efektivního chlazení. Jako řešení vzniklé situace bylo zvoleno zavedení více jader na jednotlivých procesorech a následně více procesorů v jednotlivých systémech, čímž došlo k rozložení zátěže a výraznému snížení hodnot příkonů jednotlivých částí. Při zavádění tohoto principu do oblasti IT však bylo nutné vymyslet vhodný způsob efektivního spravování takto strukturovaných HW prostředků. Vhodnou metodou je použití paralelních principů zpracování a paralelního programování (MERKLE, 2012).

Jako přiklad rozsáhlých projektů, kde je běžně využíváno paralelní zpracování, je možné uvést: výpočet předpovědi počasí, simulace pohybu částic, monitorování a vyhodnocování pohybu vesmírných těles, statistické zpracování rozsáhlých databází výsledků vědeckých prací a mnoho dalších (Ananth Grama, 2003).

Paralelní zpracování a paralelní přístup k řešení požadovaných úloh již v dnešní době není záležitostí pouze velkých vědeckých projektů, ale je součástí každodenních činností běžné populace. Tento přístup práce s procesy a vlákny je běžně uplatněn především na úrovni operačních systémů a na úrovni kódu samotných aplikací (MERKLE, 2012).

1.2 Základní pojmy

Jako další krok, po seznámení se s hlavní motivací použití paralelního přístupu, je nutné provést vymezení několika základních pojmů z této oblasti.

1.2.1 Proces

Prvním pojmem je proces. Jedná se o instanci běžícího programu v operační paměti. Tento termín je možné vysvětlit na jednoduché analogii z běžného života: pečení a receptu v kuchařce, kdy recept je analogií k programu, respektive jeho zdrojového kódu, a samotné pečení je analogií k procesu (Tanenbaum, 2001).

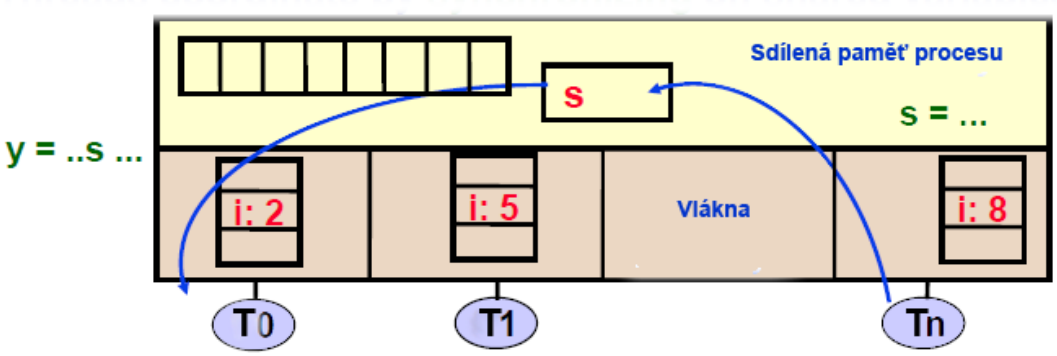
Z pohledu zkoumání paralelního programování je významnou vlastností procesu způsob práce s operační pamětí a přidělenými HW a SW prostředky. Dále je nutné se zabývat otázkami synchronizace a komunikace mezi běžícími procesy, což je podrobněji popsáno v dalších podkapitolách.

1.2.2 Vlákno

Vlákno je zjednodušeně definováno jako specifický typ potomka procesu. Je možné, aby v rámci jednoho procesu bylo spuštěno více vláken, která jsou jakousi „odlehčenou“ podobou tohoto procesu. Každé vlákno disponuje svojí vnitřní pamětí s informacemi, které jsou pro něho unikátní. Příkladem může být pozice v prováděném kódu programu, zásobník, hodnoty registrů a další. Vlákno může využívat informace z paměti

rodičovského procesu, která je pro něho a všechny jeho sourozence společná. Typicky se jedná o zdrojový kód programu (Tanenbaum, 2001).

Na následujícím obrázku (Obrázek 2) je blokově naznačena paměťová struktura několika vláken a rodičovského procesu.



Obrázek 2 - Vlákna vs. procesy

Na tomto jednoduchém příkladu je demonstrována situace, kdy každému vláknu T_0 až T_n náleží jiná hodnota lokální proměnné i a kdy mají vlákna přístup ke sdílené proměnné S rodičovského procesu (MERKLE, 2012).

1.3 Paralelizace úlohy

Paralelizaci zadané úlohy je možné v oboru informačních technologií definovat jako postup nalezení ekvivalentního kódu aplikace nebo kroků algoritmu, kde při následném spuštění jejich upravené implementace je použit více než jeden proces nebo více než jedno vlákno současně.

Tento postup není ve všech případech triviální úlohou, protože je nutné dodržet tyto základní principy:

- vhodná míra abstrakce,
- princip lokality,
- dostatečný výskyt paralelismů.

1.3.1 Vhodná míra abstrakce

Tento požadavek úzce souvisí s následujícími dvěma. V některých případech může mít dokonce rozhodující roli, protože bez vhodně abstrahovaného problému nemusí být vůbec dodržen princip lokality a nalezen dostatečný počet paralelismů.

Typickým příkladem je abstrakce modelu reálného světa, který by nebylo možné bez vypuštění některých faktorů vůbec paralelizovat. Někdy je však rozhodování o prioritě jednotlivých faktorů s ohledem na požadovanou přesnost výsledků velmi obtížné a může být otázkou empirické analýzy.

1.3.2 Princip lokality

Princip lokality je možné vysvětlit jako přihlédnutí ke skutečnosti, že přesun dat je v mnoha případech časově náročnější operací než samotné výpočetní operace s těmito daty. Tato skutečnost je dána omezenou velikostí rychlých pamětí a následným použitím hierarchického uspořádání pamětí pomalejších. Podrobněji je tato problematika popsána v podkapitole 1.6.6 vliv hierarchie pamětí.

Princip lokality je možné vysvětlit tvrzením, že je výhodnější v rámci po sobě jdoucích operací pracovat s daty, která jsou v operační paměti umístěna „blízko“ z pohledu adresy pozice, čímž dojde k efektivnímu využití cache paměti (Ananth Grama, 2003).

1.3.3 Dostatečný výskyt paralelismů

Ve skutečném světě je možné nalézt mnoho paralelismů, které nemusí být na první pohled patrné. Vhodnou abstrakcí dojde ke snížení počtu závislostí operací, případně celých objektů. S výhodou je využíván princip vypuštění nebo zjednodušení závislosti na vzdálených nebo nedůležitých objektech.

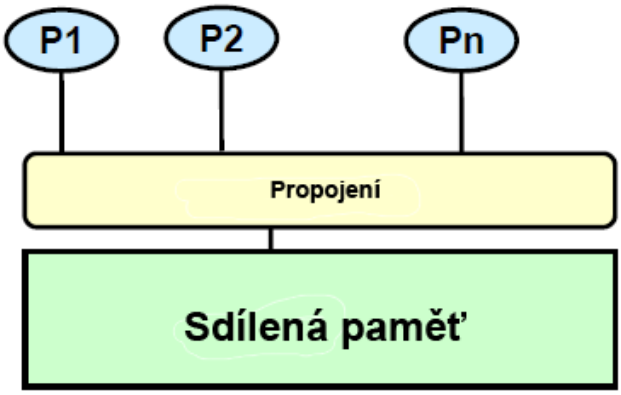
Příkladem hledání paralelismů může být řešení pohybu částic v simulátoru jejich kolizí. Pohyb částic je v takovém případě vyhodnocen po jednotlivých částicích, skupinách nebo oblastech, které je obsahují, příslušným procesem, případně vláknem (Lewis, a další, 1992).

1.4 Rozdělení paralelních přístupů

Obecně je možné způsoby paralelních přístupů k programování rozdělit na základě mnoha kritérií. Vzhledem k objektu zkoumání této práce je však nejvhodnější následující rozdělení z pohledu nakládání s operační pamětí na programovací modely:

- se sdílenou pamětí,
- s distribuovanou pamětí.

1.4.1 Programovací model se sdílenou pamětí



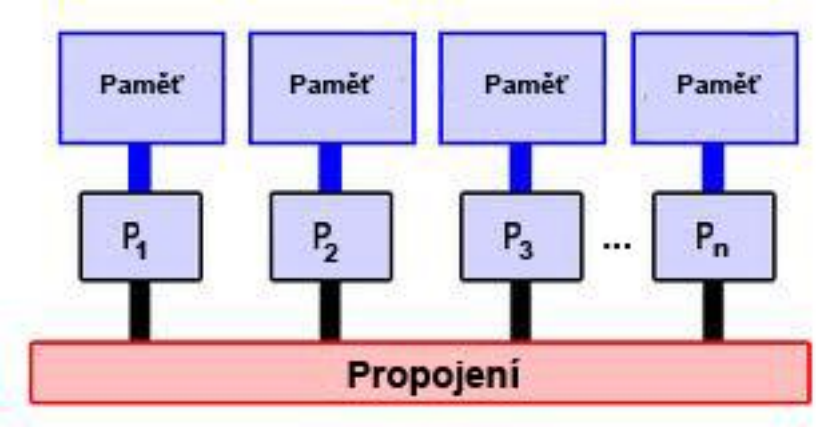
Obrázek 3 - Programovací model se sdílenou pamětí

Na obrázku 3 je schematicky zobrazen programovací model se sdílenou pamětí pro n procesů. Při tomto způsobu práce s pamětí je nezbytné vyřešit několik zásadních věcí. Jednou z nich je vzájemná komunikace procesů, která je zpravidla řešena zápisem a čtením sdílených proměnných.

Takovým řešením však vzniká nový problém, kterým je ošetření konkurenčního přístupu k těmto proměnným. Teoreticky totiž může nastat situace, kdy jeden proces čte hodnoty zde zapsané a druhý proces se pokouší na stejné místo v tento okamžik zapisovat. Dokonce je možná ještě závažnější varianta, kdy se oba procesy snaží ve stejný okamžik zapisovat do téže paměti. V praxi jsou tyto situace řešeny za využití několika principů. Jedním z nejběžnějších je použití struktur zámků, kde před každou prací s proměnnou je otestován zámek. Pokud je již zamčený, je odmítnut přístup do dané paměti a proces musí čekat na uvolnění zámku. Pokud je zámek uvolněný, je procesu přístup povolen a současně zámek zamknut. Po ukončení práce je následně opět uvolněn (Almasi, a další, 1994).

Zdrojový kód paralelní aplikace za použití modelu se sdílenou pamětí musí navíc tedy kromě běžných instrukcí ještě obsahovat ošetření přístupu do již zmíněné sdílené paměti.

1.4.2 Programovací model s distribuovanou pamětí



Obrázek 4 - Programovací model s distribuovanou pamětí

Na obrázku 4 je zjednodušený diagram programovacího modelu s distribuovanou pamětí pro n procesů. Vytváření aplikací za použití tohoto konceptu je odlišné než za použití předchozího, protože zde není jedna společná paměť, ale několik pamětí lokálních pro jednotlivé procesy, které mohou být dokonce spuštěny na serverech s geograficky rozdílným umístěním.

Z uvedeného obrázku je patrné, že v tomto případě není možné použití sdílené paměti ke vzájemné komunikaci procesů, protože jednotlivé lokální paměti nemusí být na společném HW a dokonce ani ve stejné geografické oblasti. Komunikace je v tomto programovacím modelu řešena za použití principu zasílání zpráv mezi procesy, což je podrobněji popsáno v samostatné podkapitole číslo 1.5 Komunikace a synchronizace mezi procesy.

Tento přístup tvorby aplikací má svá pozitiva i negativa. Nejvýznamnějším problémem je snížení efektivity algoritmu a celého výpočtu následkem zpoždění při čekání na odeslání, přijetí nebo přenosu zprávy (Almasi, a další, 1994).

Významným pozitivem tohoto programovacího modelu je teoreticky neomezená škálovatelnost, kde případné rozšiřování o další procesy běžící na samostatném procesoru se svojí samostatnou pamětí není otázkou limitu HW, jako je tomu v modelu se sdílenou pamětí, ale spíše otázkou maximálního možného počtu těchto procesů, který je dán způsobem paralelizace úlohy při dodržení požadované výkonnosti a efektivity nakládání s alokovanými zdroji pro tento úkol (Hill, 1990).

Zajímavým příkladem použití programování v distribuované paměti je dnes již ukončený projekt SETI, který se zabýval hledáním vyspělé inteligence ve vesmíru pomocí zachycování radiové komunikace. Ta byla následně vyhodnocena právě za použití aplikace pracující v distribuované paměti. Jako alokovaný HW pro výpočty a analýzu sloužil

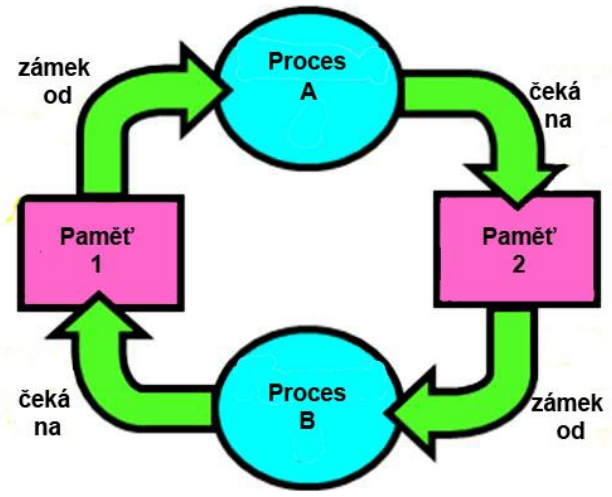
v tomto případě volný výkon dobrovolně se účastnících domácích počítačů po celém světě (SETI@home, 2013).

Typičtějším příkladem použití tohoto modelu je spouštění paralelních simulací ve výzkumném centru jako je NERSC ve Spojených státech amerických v Berkley (NERSC, 2013). Na clusteru s názvem Hooper, který je součástí tohoto výzkumného centra, byla naměřena data pro praktickou část této diplomová práce.

1.5 Komunikace a synchronizace procesů

Jak již bylo částečně popsáno výše, jednou ze zásadních vlastností paralelního programování ve sdílené i distribuované paměti je potřeba komunikace a synchronizace činnosti mezi jednotlivými procesy dané úlohy.

Komunikace mezi procesy ve sdílené paměti byla již zmíněna v podkapitole 1.4.1 Programovací model se sdílenou pamětí. Je založena na zápisu a čtení společných sdílených proměnných, což s sebou kromě již uvedeného problému konkurenčního přístupu přináší několik dalších specifik, která je nutné vyřešit. V případě čekání procesu na uvolnění zámku nad požadovanou pamětí může dojít k tzv. vyhladovění, kdy se následkem chybně nastavených pravidel konkurenčního programování dané aplikace nemusí proces nikdy dočkat. Řešením může být zavedení časového omezení, priority, pořadových čísel a dalších principů. Dalším možným problémem spojeným s přístupem více procesů do jedné paměťové oblasti je možnost uváznutí neboli deadlocku, kdy dva nebo více procesů čeká vzájemně na uvolnění více paměťových míst takovým způsobem, že není možné těmto požadavkům vyhovět bez použití speciálního algoritmu. Příkladem, který je zobrazen na obrázku 5, je situace procesů A a B, kde A zamkne oblast 1 a B zamkne oblast 2. A i B potřebují pro svoji činnost obě části paměti současně a vzájemně čekají na uvolnění té, kterou vlastní druhý proces (Lewis, a další, 1992).



Obrázek 5 - Uváznutí procesů

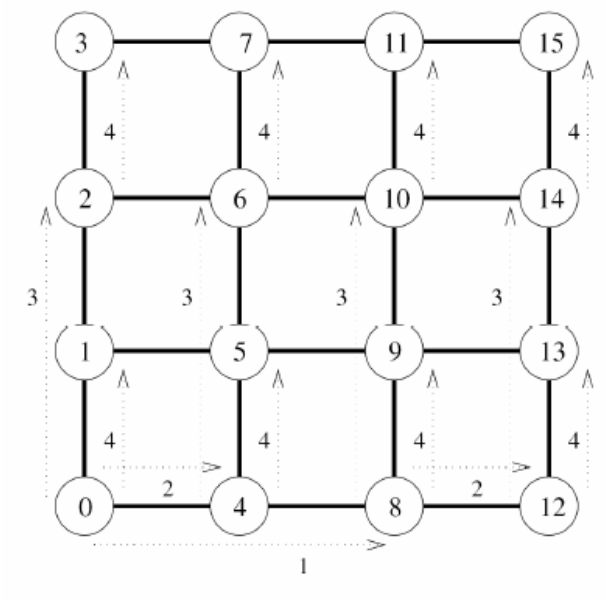
Naopak komunikace použitá v modelu s distribuovanou pamětí, která je založena na principu zasílání zpráv, nezpůsobuje tyto problémy. Může ale ovlivňovat celkovou

efektivitu aplikace, protože část celkového času běhu je použita na odesílání, přenos a přijímání zpráv. Tento důsledek použití zasílání zpráv je možné ovlivnit vhodným použitím blokujících a neblokujících volání instrukcí pro odeslání (resp. příjmu) zprávy. Zavolání blokujícího odeslání (resp. příjmu) způsobí zastavení se na této instrukci až do okamžiku jejího dokončení. Neblokující volání funguje opačným způsobem, kdy ihned po zavolání neblokující instrukce je bez ohledu na její dokončení pokračováno ve vykonání následující instrukce. Tento princip byl s výhodou použit v praktické části práce při implementaci paralelní Gaussovy eliminace, kde každé odeslání zprávy bylo voláno neblokujícím způsobem, což zvyšuje efektivitu aplikace, a každý příjem zprávy byl volán blokujícím způsobem, což zvyšuje bezpečnost kódu (Lewis, a další, 1992).

Činnost vláken a procesů je z důvodu zachování stejných výsledků a chování aplikace jako před paralelizací kódu nutné synchronizovat. K tomu se jak v modelu se sdílenou, tak v modelu s distribuovanou pamětí používá bariér, jejichž princip je možné popsat následujícím způsobem: Vlákna nebo procesy, které zavolají instrukci bariéry, musí na tomto místě v kódu zastavit a čekat. V okamžiku, kdy všechny zavolají tuto instrukci, je možné pokračovat dál na následující příkaz vlákna nebo procesu. Tento jednoduchý princip má však svá úskalí. Nejproblematičtějším z nich je možnost uváznutí celé aplikace, které může být způsobeno deadlockem několika vláken nebo procesů před bariérou a následným čekáním všech ostatních na blokujícím volání instrukce bariéry. Toto riziko je v některých paralelních knihovnách minimalizováno možností třídit vlákna a procesy do logicky souvisejících skupin, nad kterými je možné volat blokující instrukce bariéry (Ananth Grama, 2003).

V následujících odstavcích jsou ještě stručně popsány některé specifické případy zasílání zpráv mezi procesy v distribuované paměti. Použití těchto principů, které jsou ale závislé na využívané paralelní knihovně, může výrazným způsobem ovlivnit celkovou efektivitu napsaného kódu tím, že je odeslání nebo příjem zpráv provedeno ve vhodnějším pořadí, než se může na první pohled zdát logické.

Typickým případem je například zaslání tzv. broadcast zprávy, což je zaslání stejného obsahu od jednoho procesu všem ve skupině. Tuto úlohu je možné triviálně vyřešit jednoduchým cyklickým zasíláním všem procesům. Trochu inteligentněji je možné provést broadcast odesílání za použití speciální instrukce k tomu určené, která na základě charakteru topologie propojení HW rozhodne o vhodnějším pořadí. Pro názornější vysvětlení problematiky je na následujícím obrázku uveden konkrétní příklad použití implementace této funkce v knihovně MPI pro 16 nodů v gridové topologii (Ananth Grama, 2003).



Obrázek 6 - Broadcast odeslání zprávy pomocí knihovny MPI v gridové topologii

Z obrázku je patrné, že místo postupného odeslání zprávy patnácti procesům v cyklu byla zvolena výhodnější strategie vzhledem k dané topologii propojení nodů, která zabrala pouze čtyři kroky. V prvním a druhém kroku je zpráva distribuována v rámci prvního řádku, ve třetím kroku je paralelně odeslána všem nodům třetího řádku a v posledním čtvrtém kroku je paralelně odeslána do druhého a čtvrtého řádku (Cormen, a další, 1990).

Dalším specifickým případem je redukce. Jedná se o zaslání zpráv od všech procesů ve skupině jednomu konkrétnímu procesu. Postup je přesně opačný než u uvedeného broadcast odesílání v předchozím odstavci.

Existuje ještě celá řada dalších dnes již standardizovaných instrukcí pro efektivnější komunikaci mezi procesy v distribuované paměti, které není nutné podrobněji popisovat, protože nebyly využity v praktické části této diplomové práce. Zde je uveden pouze informativní výpis některých z nich:

- broadcast odeslání od všech procesů všem procesům,
- redukce zpráv všech procesů všem procesům,
- postupné načtení zpráv jedním procesem od všech procesů v daném pořadí,
- postupné odeslání zpráv jedním procesem všem procesům v přesně daném pořadí,
- a další (Cormen, a další, 1990).

1.6 Hodnocení paralelních a sériových algoritmů

Pro objektivní porovnávání implementací paralelních a sériových algoritmů je nezbytné definovat a podrobněji vysvětlit některé pojmy z oblasti analytického modelování paralelních systémů a z oboru hodnocení výkonnostních metrik těchto systémů.

1.6.1 Čas běhu vs. čas neefektivity

Základním hodnotícím ukazatelem zkoumaného algoritmu je čas běhu. Ten je možné rozdělit na dva základní typy: sekvenční a paralelní. Ještě před podrobnějším vysvětlením těchto pojmů je nutné uvést jednu zásadní poznámku a to, že čas běhu je veličina závislá na platformě a HW, na kterém je úloha spuštěna. Z tohoto jasně plyne, že pro jakékoliv srovnávání je nutné stanovit vždy stejné podmínky pro všechny běhy měření.

Sériový čas běhu, který se značí T_s, je definován jako čas naměřený mezi začátkem a koncem běhu algoritmu na sériovém, tedy jednoprocesorovém, stroji. Při zanedbání vlivu hierarchie pamětí, který je popsán v podkapitole 1.6.6, je možné tento čas teoreticky vypočítat jako funkci složitosti algoritmu, velikosti vstupu a maximálního výkonu HW.

Paralelní čas, který je v literatuře značen T_p, je možné definovat jako čas naměřený mezi začátkem činnosti prvního procesu a koncem činnosti posledního procesu dané paralelní úlohy. Hodnota této veličiny je značně ovlivněna zpožděními, která jsou dána vzájemnou interakcí a čekání procesů při komunikaci a synchronizaci. Dále může být ještě ovlivněna pomocnými výpočty, jenž se v původní sériové verzi algoritmu nevyskytují a jenž jsou výsledkem složitější paralelizace některých algoritmů.

Na základě předchozí definice je možné vysvětlit pojem celkový čas paralelního běhu algoritmu, který je dán jako souhrn časů činnosti všech procesů daného úkolu. Tato veličina je označena jako T_{all} a je možné ji vypočítat dle následujícího vztahu:

$$T_{all} = T_p * p$$

kde T_{all} – je celkový čas paralelního běhu algoritmu,
T_p – je paralelní čas běhu,
p – je počet použitých procesorů.

Za použití stejného značení veličin je možné definovat ještě celkový čas neefektivity, který je dán jako rozdíl celkového času běhu algoritmu a sériového času běhu a který je značen T_O. Hodnota této veličiny v sobě zahrnuje již výše zmíněný vliv komunikace, synchronizace a pomocných výpočtů. Je možné ji vypočítat za použití následujícího vztahu:

$$T_O = T_p * p - T_S$$

kde T_O – je celkový čas neefektivity,
T_S – je sériový čas běhu.

1.6.2 Zrychlení

Tento pojem, který se v odborné literatuře značí S, je v podstatě vyjádřením benefitu získaného díky použití paralelního přístupu při řešení zadané úlohy. Jedná se o poměr sériového a paralelního času běhu programu, neboli o poměr času, který je nezbytný pro vyřešení zadané úlohy na jednoprocesorovém systému a času, který je

naměřen při použití paralelního programování na víceprocesorovém systému. Hodnotu této veličiny je možné získat ze vztahu:

$$S = \frac{T_S}{T_p}$$

kde S – je zrychlení,
T_S – je sériový čas běhu,
T_p – je paralelní čas běhu (Nicol, a další, 1988).

1.6.3 Efektivita

Dalším významným pojmem, který je nezbytné definovat, je efektivita. Jde v podstatě o vyjádření míry efektivního využití procesorů. V literatuře je tato veličina označena jako E a její hodnotu, která se pohybuje v rozmezí intervalu $< 0;\ 1 >$, je možné vypočítat dle tohoto vzorce:

$$E = \frac{S}{p}$$

kde S – je zrychlení,
p – je počet použitých procesorů.

Díky hierarchii pamětí, komunikaci a všech dalších již několikrát zmíněných vlivů, je prakticky nemožné dosáhnout hodnoty efektivity rovné jedné na víceprocesorovém systému (Grama, a další, 1993).

1.6.4 Optimální „cena“

Cena v kontextu paralelního programování nemá prakticky nic společného s pojmem, který je definován v oboru ekonomie. V oblasti zkoumání této práce se jedná o vyjádření doby sériového nebo celkového paralelního běhu algoritmu a tím i jeho sériové nebo celkové paralelní „ceny“. Dosazením vztahu pro výpočet zrychlení S do rovnice pro výpočet efektivity E je dán následující výsledek:

$$E = \frac{\frac{T_S}{T_p}}{p}$$

Po úpravě je získán tento vztah:

$$E = \frac{T_S}{p * T_p}$$

Cena použité paralelní implementace algoritmu může být označena za optimální, pokud platí tento vztah:

$$E = O(1) = \frac{T_S}{p * T_p}$$

Čímž je myšleno, že cena řešení na sériovém stroji je asymptoticky identická s celkovou cenou řešení na paralelním stroji. Výpočtem tohoto poměru je možné posoudit smysluplnost a ekonomičnost použití paralelní verze algoritmu pro danou úlohu (Eager, a další, 1989).

1.6.5 Vliv granularity a škálovatelnosti

Z pohledu zkoumání efektivity a vlastního porovnávání algoritmů je nutné také definovat vliv granularity a již zmíněné škálovatelnosti, která je na jednu stranu významnou předností paralelních systémů v distribuované paměti a na stranu druhou faktorem, který může významným způsobem snížit efektivitu hodnocené implementace zkoumaného algoritmu.

Z následujícího vztahu pro výpočet efektivity po dosazení rovnice pro výpočet zrychlení je možné odvodit tento vliv:

$$E = \frac{S}{p} = \frac{T_S}{p * T_p}$$

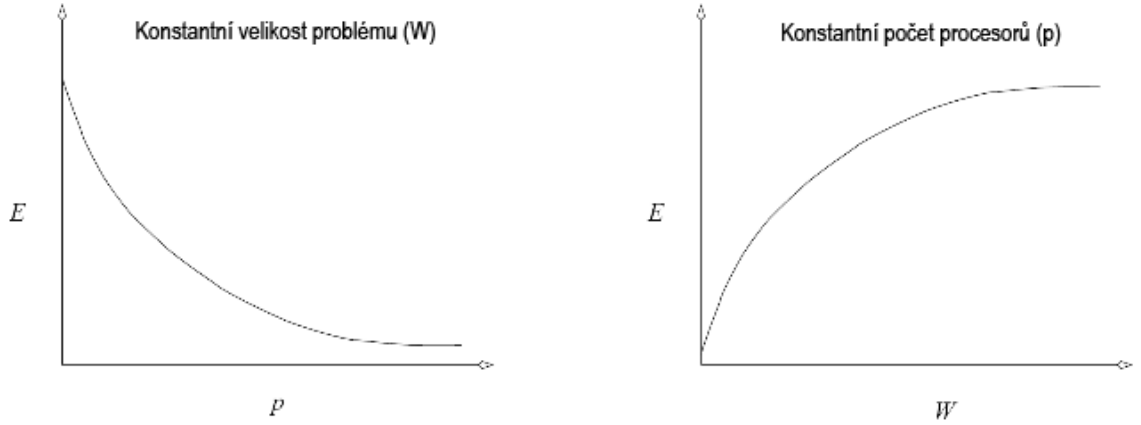
Dosazením rovnice pro výpočet celkového času neefektivity je E vyjádřeno takto:

$$E = \frac{T_S}{T_O + T_S} = \frac{1}{1 + \frac{T_O}{T_S}}$$

Z výše uvedeného je patrné, že s nárůstem času T_O, který narůstá s počtem použitých procesorů, dojde ke snížení efektivity. Tento vliv škálovatelnosti paralelního systému se dá vysvětlit tak, že použití více procesorů při zachování velikosti řešeného problému způsobí nárůst času potřebného pro meziprocesorovou komunikaci a synchronizaci jejich činností (Hill, 1990).

Poměr $\frac{T_O}{T_S}$ je možné následně použít jako koeficient při stanovení vhodného počtu použitých procesorů pro danou velikost úlohy. Tento postup však přesahuje téma a rozsah této diplomové práce. V praktické části byl počet stanoven experimentální metodou.

Obdobným způsobem je možné definovat vliv nárůstu velikosti řešeného problému při zachování počtu použitých procesorů. Touto problematikou se podrobněji zabývá (Ananth Grama, 2003). Na následujícím obrázku, který uvádí MERKLE (2013), je patrné, že vlevo je naznačen příklad průběhu efektivity E pro konstantní velikost problému n a rostoucí počet procesorů p, a vpravo pro konstantní počet použitých procesorů p a rostoucí velikost řešeného problému n, kterým je v tomto případě paralelizace sčítání n čísel v distribuované paměti.



Obrázek 7 - Vliv škálovatelnosti paralelního systému

kde E – je efektivita,
p – je počet použitých procesorů,
W – je velikost řešené úlohy.

1.6.6 Vliv hierarchie pamětí

Dalším významným pojmem, který je nutné vysvětlit a vzít v úvahu při porovnávání sériových a paralelních algoritmů, je vliv hierarchie pamětí použitého HW. Z pohledu zkoumání této práce je možné tento aspekt zjednodušit a zabývat se pouze jednoprocesorovými systémy a následným zobecněním pro systémy víceprocesorové.

Většina běžných neoptimalizovaných aplikací pracuje s efektivitou cca deset až dvacet procent. Převážná část efektivity je totiž ztracena přesuny a dalšími operacemi v paměťovém systému použitého HW, protože tyto operace jsou časově mnohem náročnější než operace aritmetické a logické, které provádí procesor. Tento vliv je možné demonstrovat na jednoduchém příkladu příkazu A + B = C. Řešení této triviální úlohy může být ve skutečnosti složeno z několika kroků:

1. načtení adresy B do registru R1,
2. načtení adresy A do registru R2,
3. operace sečtení obsahu registrů R1 a R2,
4. uložení výsledku do registru R3
5. a zpětný zápis registru R3 na adresu C.

Na uvedeném příkladu je patrné, že triviální instrukce sečtení, která zabere řádově několik nanosekund, je značně zpomalena třemi velmi časově náročnými vstupně-výstupními operacemi. Míra zpomalení je dána typem paměti, ze které byla data načtena nebo do které byla zapsána. Orientační přehled některých pamětí včetně řádových hodnot jejich rychlostí a velikostí je uveden v následující tabulce.

Tabulka 1 - Přehled typů pamětí

Typ paměti	Rychlost přístupu (čas)	Velikost
Registry procesoru	1 ns	kB
L1 cache na čipu	1 ns	kB
L2 cache (SRAM)	10 ns	MB
Operační paměť (DRAM)	100 ns	GB
Externí paměť (HD)	10 ms	TB
Síťové diskové pole (HD, páska)	10 sec	PB

Z uvedených parametrů jsou patrné velmi nízké kapacity rychlých pamětí, které jsou dány technickou a finanční náročností výroby a limity použitých materiálů. Tento problém je v praxi vyřešen použitím SW a HW optimalizačních technik, jejichž cílem je zvýšení pravděpodobnosti použití rychlejších pamětí.

Jedním z principů, který je použit především v případě cache pamětí, je postupné přesouvání těch nejpoužívanějších dat do co nejrychlejších pamětí a postupné odsouvání těch méně používaných dat do pamětí pomalejších. Tento algoritmus musí být optimalizován takovým způsobem, aby čas použitý k přesunu dat nepřevýšil výhodu získanou efektivnějším využitím rychlejších paměťových oblastí (MERKLE, 2012).

Dalším možným řešením je implementace algoritmů s přísným dodržením principu lokality, který byl popsán v podkapitole 1.3.2, čímž dojde k efektivnějším přenosům dat ve skupinách bez zbytečného dalšího prohledávání pomalé paměti a ke zvýšení parametru cache hit, který značí úspěšnost při hledání požadovaných dat v rychlé cache paměti. Na tomto principu je založen jeden ze způsobů optimalizace kódu, kdy jsou cykly s velkým počtem iterací rozděleny na vnořenou strukturu několika hierarchických cyklů. Tento princip, i když se může zdát na první pohled neefektivní, je možné po stanovení vhodného počtu hierarchií a iterací vnořených cyklů na základě velikosti zpracovávaných dat a velikosti cache pamětí použitého HW využit k zefektivnění kódu aplikace z pohledu principu lokality (Ananth Grama, 2003).

Specifickým a na použitém HW závislým řešením je změna pořadí a náhrada běžných příkazů v kódu speciálními instrukcemi s možností paralelního zpracování. Typickým příkladem je možnost provedení několika násobení současně v násobící jednotce procesoru.

Některé z výše uvedených a dalších technik řeší automaticky procesor, řadiče pamětí, kompilátory a operační systémy s určitým stupněm zlepšení efektivity. Ve specifických případech, především u úloh s velkými nároky na výkon a přesnost výsledků, je výhodné nepoužít automaticky nabízených optimalizačních služeb, ale naimplementovat alespoň část některých technik ručně na míru daného problému. V praxi je s výhodou použita ještě celé řada dalších optimalizačních technik, které zde nebyly vyjmenovány.

Všechny doposud popsané techniky a principy se týkaly zjednodušeného systému s jedním procesorem. V případě paralelního systému s distribuovanou pamětí[1] je situace složitější z důvodu nesouvislé paměti a vzájemné meziprocesorové komunikace. Teoreticky totiž může nastat situace, kdy proces A běžící na nodu x nemůže použít výhody hromadného načtení a zpracování dat z paměti, protože v okamžiku, kdy by mohl tuto činnost provést, je část těchto dat v lokální paměti procesu B, který běží na nodu y. Takovým situacím je důležité předcházet důslednou analýzou již v době návrhu algoritmu, protože mohou výrazně snížit celkovou efektivitu. V některých specifických případech je část dat načtena v průběhu výpočtu do paměti procesu A redundantně dopředu. Režie plynoucí z této redundance by však neměla převýšit výhodu získanou z možnosti použití principu lokality na systému s nesouvislou distribuovanou pamětí.

[1] Z pohledu zkoumání této práce není významné popisovat tento princip také pro paralelní systémy se sdílenou pamětí.

2 Gaussova eliminace

2.1 Základní popis a význam metody

Gaussova eliminace je konečná metoda řešení soustavy lineárních algebraických rovnic. Je nazývána konečnou, protože vede k přesnému řešení v konečně mnoha krocích (Vitásek, 1987).

Problém vypočtení kořenů soustavy lineárních algebraických rovnic je součástí mnoha dalších technických, matematických, vědeckých, ekonomických a jiných úloh. Z tohoto důvodu je nezbytné mít efektivní a hlavně konečný způsob jejich vyřešení, kterým může být již zmíněná metoda. Příkladem praktického využití řešení soustav rovnic mohou být vědecké studie a experimenty, simulace a předpověď počasí, simulace ve výrobních a vývojových odvětvích a mnoho dalších.

Gaussovu eliminační metodu je možné použít také v průběhu výpočtu inverzní matice nebo determinantu. Tím se značně rozšiřují možnosti jejího praktického využití. Typickým příkladem mohou být oblasti robotiky a zpracování obrazu, kde jsou maticové počty běžnou nepostradatelnou součástí velké části výpočtů.

2.2 Obecný postup eliminace

Gaussova eliminační metoda řeší soustavu lineárních algebraických rovnic, která je dána následujícím předpisem:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \ldots + a_{1,n-1}x_{n-1} &= b_1 \\ &\ldots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned}$$

Tento vztah je možné přepsat v maticové podobě na:

$$Ax = b$$

kde A – je matice koeficientů $a_{i,j} = A[i,j]$ o rozměrech $n\ x\ n$,
b – je vektor $[b_0, b_1, \ldots, b_{n-1}]^T$ o rozměrech $n\ x\ 1$,
x – je vektor hledaného řešení $[x_0, x_1, \ldots, x_{n-1}]^T$ o rozměrech $n\ x\ 1$.

Gaussova eliminace pracuje obvykle ve dvou krocích. Prvním je použití série algebraických operací nad řádky matice A a vektorů x a b, jejichž cílem je redukce matice A na horní trojúhelníkovou formu, která je dána tímto algebraickým předpisem:

$$\begin{aligned} x_0 + u_{0,1}x_1 + u_{0,2}x_2 + \cdots + u_{0,n-1}x_{n-1} &= y_0 \\ x_1 + u_{1,2}x_2 + \cdots + u_{1,n-1}x_{n-1} &= y_1 \\ &\ldots \\ x_{n-1} &= y_{n-1} \end{aligned}$$

Tento předpis může být zapsán v maticové podobě takto:

$$Ux = y$$

kde U – je horní trojúhelníková matice,
$U[i,j] = 0$ pro $i > j$,
$U[i,j] = 1$ pro $0 \leq i < n$,
jinak $U[i,j] = u[i,j]$.

Po provedení těchto úprav je možné dle Frobeniovy věty určit, zda existují reálná řešení této soustavy rovnic. Pokud je hodnost výsledné matice po úpravách jiná než hodnost upravené rozšířené matice o pravou stranu zadaných rovnic, pak nemá soustava žádné řešení. Naopak jsou-li tyto hodnosti rovny a jsou současně rovny počtu neznámých, mají rovnice právě jedno řešení. Poslední možností je nekonečně mnoho řešení, které je možné rozpoznat z uvedené rovnosti hodností matic za podmínky, že je tato hodnost menší než počet neznámých (Vitásek, 1987).

Druhým krokem Gaussovy eliminační metody je zpětná substituce, která je podrobněji popsána v následující podkapitole.

2.3 Zpětná substituce

Zpětná substituce je druhým krokem Gaussovy eliminace, při kterém jsou z horní trojúhelníkové matice vypočteny kořeny řešených rovnic.

Tento postup je proveden v opačném pořadí od $x[n-1]$ do $x[0]$. Výsledkem každé iterace je jeden kořen x, přičemž $x[\mathrm{n}-1]$ je již přímo vypočten po úpravě matice A na horní trojúhelníkovou matici.

Zpětnou substituci je možné popsat jako cyklické dosazování vyřešených kořenů do předchozích rovnic a následné vypočtení jejich kořenů, počínaje x[n-2]. Podrobný algoritmus tohoto postupu je uveden v podkapitole 3.3.

3 Implementace Gaussovy eliminace

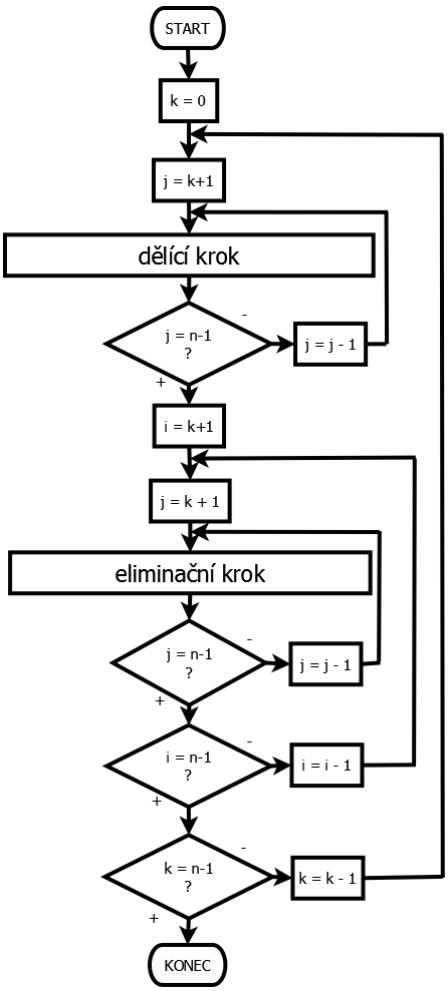
Existuje celá řada implementací Gaussovy eliminační metody. V následující podkapitole je popsána „nejtriviálnější“ sériová metoda, která je založena na prostém cyklickém upravení matice A a vektorů b a x.

3.1 Sériová metoda implementace

3.1.1 Popis algoritmu

Základní sériová metoda implementace Gaussovy eliminační metody je založena na principu třech vnořených cyklů. Existuje několik variant těchto implementací, které se liší pořadím jejich vnoření. V následujících odstavcích bude vysvětlena varianta, která bude v další podkapitole použita při paralelizaci.

Gaussova eliminační metoda přepočítává soustavu lineárních algebraických rovnic, která je dána předpisem $Ax = b$, na systém s horní trojúhelníkovou maticí, který je dán předpisem $Ux = y$. Tento postup již byl podrobněji popsán v podkapitole 2.2. Na následujícím obrázku je schematicky naznačen princip první části sériové implementace Gaussovy eliminační metody (Foster, a další, 1995).



Obrázek 8 - Algoritmus sériové implementace Gaussovy eliminace

Na uvedeném diagramu a v dalším popisu bude vysvětlena varianta sériové implementace, která v průběhu přepisuje původní matici A výslednou maticí U.

Vybranou součást Gaussovy eliminační metody, kterou je úprava matice A na horní trojúhelníkovou matici U, je možné logicky rozdělit na dvě části. První z nich je tzv. dělící krok, který je možné za použití značení z předchozího obrázku popsat takto:

$$U\,[k,j] = \frac{A[k,j]}{A\,[k,k]}$$

kde U – je výsledná trojúhelníková matice,
A – je matice koeficientů $a_{i,j} = A[i,j]$ o rozměrech $n\ x\ n$,
j, k – jsou koeficienty dle značení na předchozím diagramu.

Slovně je možné tento postup popsat jako vydělení prvků k-tého řádku matice A prvkem $A\,[k,k]$, který leží na hlavní diagonále této matice, kde současně platí, že dělené prvky leží napravo od dělícího prvku $A\,[k,k]$. Tento dělící krok je možné provést pouze za předpokladu, že matice A je silně regulární, z čehož vyplývá, že dělící prvek $A\,[k,k]$ nemůže být roven nule.

Obdobným způsobem je možné popsat tento postup pro vektor y:

$$y[k] = \frac{b[k]}{A[k,k]}$$

Druhou částí je tzv. eliminační krok, který je definován při zachování značení z popisovaného obrázku takto:

$$U[i,j] = A[i,j] - A[i,k] * A[k,j]$$

kde i – je koeficient dle značení na uvedeném diagramu.

Eliminační krok je popsán jako postupné odečítání násobků prvku i-té řady, který umístěn nalevo vedle hlavní diagonály matice A, a j-tého prvku k-té řady od prvku i-té řady, který je pod tímto prvkem umístěn rovněž jako j-tý v pořadí.

Pro vektory b a y je možné tento postup definovat jako:

$$b[i] = b[i] - A[i,k] * y[k]$$

Z uvedeného popisu a diagramu je možné vypozorovat, že výpočet v k-té iteraci vnějšího cyklu je „aktivní“ pouze pro oblast matice A se souřadnicemi většími než k. Z toho plyne, že výpočet aktivně pracuje pouze s oblastí o velikosti $(n-k) * (n-k)$. Tento důsledek sériové implementace je značně limitujícím faktorem z pohledu paralelizace a celkové efektivity paralelní implementace.

3.1.2 Stanovení složitosti algoritmu

Na základě znalosti uvedené implementace sériového algoritmu a empirické analýzy je možné stanovit asymptotickou složitost.

V průběhu dělících kroků je provedeno dělení s přibližně polovinou prvků z množiny $n * n$. Z čehož plyne, že celková náročnost těchto operací za předpokladu, že složitost jedné operace je rovna jedné, je dána jako:

$$W_d = \frac{n * n}{2}$$

Kde W_d – je složitost dělícího kroku,
n – je rozměr matice A.

Počet eliminací a násobení v eliminačním kroku je definován, jak uvádí (Ananth Grama, 2003), jako:

$$W_e = W_n = \frac{n^3}{3} - \frac{n^2}{2}$$

Kde W_e – je počet eliminací,
W_n – je počet násobení.

Výslednou složitost je následně možné určit jako součet uvedených složitostí dělícího a eliminačního kroku takto:

$$W = W_d + W_e + W_n$$

$$W = \frac{n^2}{2} + 2 * \left(\frac{n^3}{3} - \frac{n^2}{2}\right)$$

$$W = 2 * \frac{n^3}{3} - \frac{n^2}{2} \approx 2 * \frac{n^3}{3}$$

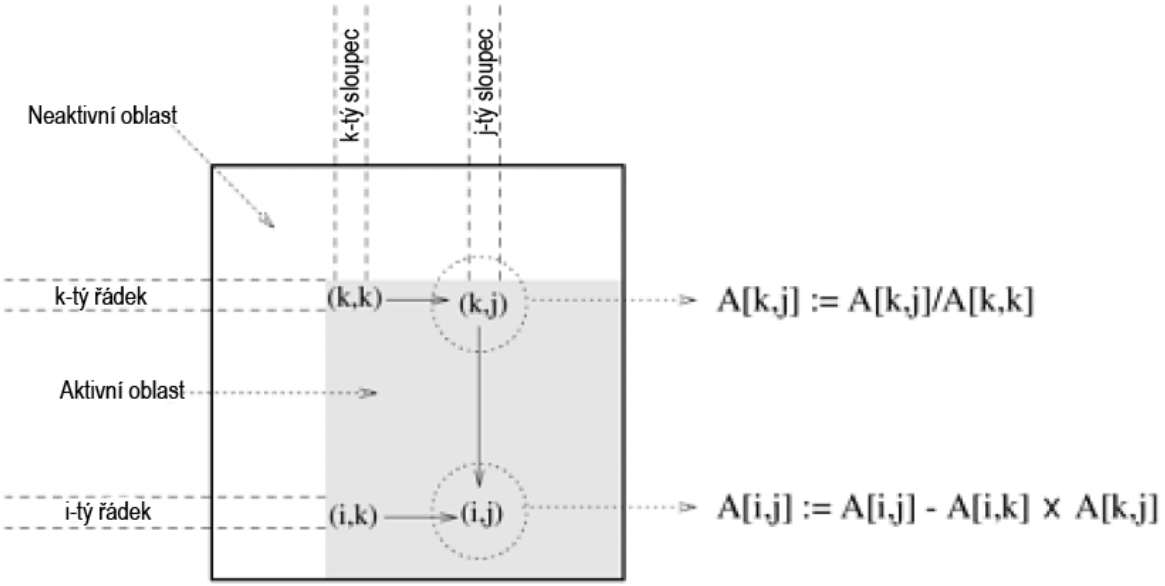
Kde W – je celková složitost algoritmu.

3.2 Paralelizace a paralelní implementace

Paralelizace, která již byla definována v podkapitole 1.3, není v případě Gaussovy eliminační metody triviální úlohou. Největšími problémy, které je nutné v průběhu tohoto procesu, vyřešit jsou:

- nalezení dostatečného počtu paralelizmů,
- efektivní použití principu lokality,
- ošetření již zmíněné „neaktivní“ oblasti matice A v průběhu výpočtu.

Na níže uvedeném obrázku je schematicky naznačen problém s aktivní a neaktivní oblastí matice A v průběhu výpočtu.



Obrázek 9 - Problém s neaktivní oblastí v průběhu Gaussovy eliminace

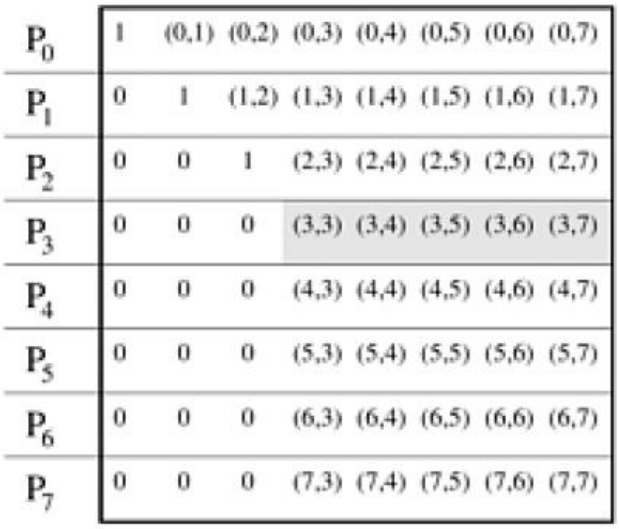
Toto chování, které je typické pro sériovou implementaci s vnořenými cykly, je velkou komplikací v průběhu paralelizace, protože značným způsobem snižuje počet paralelizmů. V podkapitole 3.2.2 je popsán způsob, jakým byl tento problém částečně vyřešen v rámci drobné modifikace zvolené paralelní implementace, která byla použita při měření dat pro tuto práci.

3.2.1 1-D dělení

V praktické části práce byla zvolena pro paralelizaci Gaussovy eliminace metoda s názvem 1-D dělení, kterou je možné zjednodušeně popsat jako jednorozměrné rozdělení použitých vstupních dat a jejich následné paralelní zpracování.

Nalezení takového rozdělení v případě sériové Gaussovy eliminační metody není vůbec triviální úlohou, protože většina příkazů sériového algoritmu je značně závislých na pořadí provedení, což není v případě paralelizace žádoucí vlastností.

Na následujících obrázcích je detailněji popsán a vysvětlen princip použitého 1-D dělení na konkrétním případu rozdělení matice A mezi 8 procesů (Ananth Grama, 2003). Rozměr matice a počet procesů na obrázku byl zvolen jen pro ilustraci. V průběhu experimentů byly použity hodnoty o několik řádů vyšší.



Obrázek 10 - 1-D rozdělení matice A

Na obrázku 10 je schematicky naznačena situace po čtvrté iteraci vnějšího cyklu, který je na obrázku 8 řízen proměnnou k. V tento okamžik je již hlavní diagonála matice A zaplněna na prvních třech řádcích jedničkami a na pozicích prvních třech sloupců pod diagonálou nulami. Je zde rovněž naznačen princip zmíněného 1-D dělení takovým způsobem, že každému procesu P_0 až P_7 byl přiřazen právě jeden řádek matice.

Rozdělení matice mezi procesy se provádí tak, že každému procesu je přiřazeno několik řádků matice A. Tento počet je dán výsledkem po celočíselném dělení rozměru matice n počtem procesů p. O tuto činnost se stará nultý proces, který má na začátku výpočtu k dispozici všechna data, takovým způsobem, že provede rozdělení matice na skupiny řádků dle vypočítaných velikostí a v daném pořadí je zašle příslušným procesům. Z tohoto chování algoritmu vyplývá podmínka, která limituje jeho možnosti:

$$p \leq n$$

kde p – je počet procesů,
n – je rozměr čtvercové matice A.

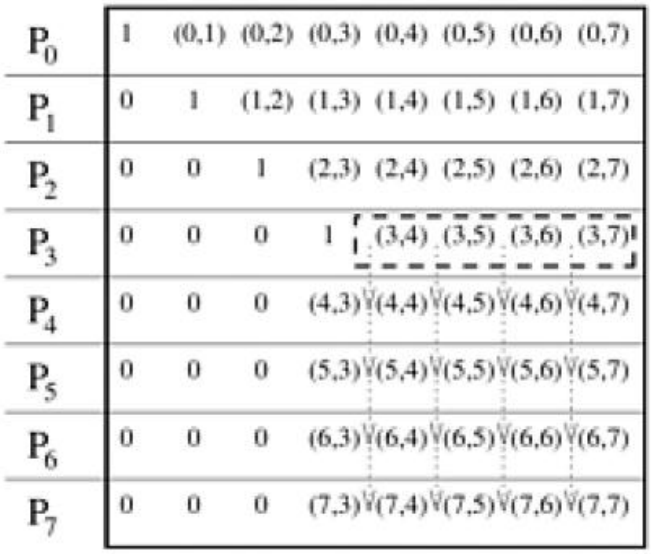
Dalším omezením, které se týká počtu použitých procesů, je skutečnost, že s rostoucím rozdělením matice, a tedy i s rostoucím počtem procesů dochází postupně k větším ztrátám efektivity, jež jsou dány zvýšenou potřebou komunikace a synchronizace mezi procesy.

Zbytkové řádky po tomto rozdělení mohou být ošetřeny různými způsoby. Jedno konkrétní řešení, které bylo použito při implementaci paralelní Gaussovy eliminace, je popsáno v následující podkapitole 3.2.2 Zvolená modifikace 1-D dělení.

V rámci k-té iterace algoritmu provede proces, který je vlastníkem k-tého řádku matice A, dělící krok pro tento řádek stejným způsobem, jako je popsán v podkapitole 3.1.1 pro sériovou implementaci. Důležitou poznámkou pro tuto fázi výpočtu je, že v tento

okamžik může být provedena tato operace pouze pro jeden řádek. Prakticky je toto ošetřeno použitím instrukce bariéry, což znamená, že všechny procesy, které v daný okamžik neprovádí dělící krok, musí na tomto místě čekat na dokončení činnosti procesu, který je vlastníkem aktuálně řešeného řádku. Tato vlastnost značným způsobem ovlivňuje celkovou efektivitu a je příčinou nárůstu času neefektivity T_O. V nejhorším případě je totiž teoreticky možné, že dojde k pozastavení činnosti všech procesů kromě již právě zmíněného vlastníka k-tého řádku.

Výsledná data po dělícím kroku je nutné rozdistribuovat všem procesům, jež jsou vlastníky submatic matice A, které leží níže než v tento okamžik řešený řádek.



Obrázek 11 - Distribuce výsledků dělícího kroku

Na obrázku 11 je uveden příklad, kdy proces P_3 dokončil dělící krok pro čtvrtý řádek matice A. Výsledná data je nyní nutné rozdistribuovat všem procesům P_4 až P_7, jež jsou vlastníky submatic položených níže než aktuálně řešený řádek. Distribuce může být provedena za použití běžného cyklického odeslání nebo za použití speciálních instrukcí, jež byly popsány na konci podkapitoly 1.5. V implementaci, která byla použita v praktické části práce, byla zvolena varianta cyklického odeslání za použití optimalizace pomocí neblokujících volání. Princip této optimalizace je rovněž detailněji popsán v podkapitole 1.5.

V okamžiku, kdy procesy, které spadají do aktivní oblasti výpočtu[2], přijmou zaslané výsledky dělícího kroku a zahájí výpočty, jež jsou součástí eliminačního kroku Gaussovy eliminace. Tento postup je schematicky naznačen na následujícím obrázku.

[2] Problém s aktivní a neaktivní oblastí matice A je podrobněji popsán na začátku podkapitoly 3.2.

P_0	1	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
P_1	0	1	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
P_2	0	0	1	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
P_3	0	0	0	1	(3,4)	(3,5)	(3,6)	(3,7)
P_4	0	0	0	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
P_5	0	0	0	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
P_6	0	0	0	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
P_7	0	0	0	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Obrázek 12 - Eliminační krok paralelní implementace Gaussovy eliminace

Na obrázku 12 je zobrazen příklad situace, kdy procesy P_4 až P_7 obdrželi výsledky dělícího kroku, který provedl proces P_3. V tento okamžik mohou P_4 až P_7 zahájit výpočty spadající do eliminačního kroku pro řádky matice A, jejichž jsou vlastníkem.

Situace, kdy skupina procesů zahájí paralelně již zmíněné výpočty, je jediným avšak významným benefitem paralelní implementace za použití metody 1-D dělení.

Z výše popsaných principů paralelní implementace je patrné, že postupujícím výpočtem dojde vlivem problému s aktivní a neaktivní oblastí matice A ke snižování počtu procesů, jež se podílí na průběhu tohoto výpočtu. Tato tendence významným způsobem ovlivní celkovou efektivitu E a zrychlení S.

Protichůdným jevem, který se projeví v průběhu postupujících iterací procházejících přes jednotlivé řádky a který částečně kompenzuje výše uvedené negativum, je pokles celkového času neefektivity T_O následkem snižujícího se počtu procesů, které se účastní výpočtu. Podrobněji byla tato problematika popsána v podkapitole 1.6.5, jež se zabývá otázkami škálovatelnosti a granularity.

Výsledkem uvedených tendencí jsou nestejné výpočetní časy jednotlivých procesů, což prakticky znamená, že procesy s nižším pořadovým číslem, které zpracovávají výše položené řádky matice A, využijí menší část svého celkového času běhu pro výpočet než procesy s vyšším pořadovým číslem, které se delší dobu nacházejí v aktivní oblasti výpočtu.

3.2.2 Modifikace 1-D dělení

Z výše popsaných negativ vychází jeden optimalizační princip, který byl použit při implementaci zkoumaného paralelního algoritmu. Jedná se o vhodnější využití okrajových řádků matice, jež vzniknou jako zbytková část po jejím rozdělení. Tato krajní submatice je výsledkem faktu, že ve většině případů není rozměr matice celočíselně dělitelný počtem procesů. Při použití klasické implementace 1-D dělení by byla tato část přiřazena poslednímu procesu, který je ovšem z pohledu času věnovaného výpočtu nejaktivnější a zbytečně by došlo k prodloužení celkového paralelního času běhu T_P. V modifikované verzi 1-D dělení je však zbytková část matice přiřazena procesům s nejnižším pořadovým číslem, které by se jinak v průběhu výpočtu staly neaktivními následkem problému s neaktivní oblastí matice A příliš brzy.

Z tohoto popsaného principu vychází další možnost modifikace běžného 1-D dělení, která nebyla v experimentální části práce použita pro svou vysokou náročnost implementace, ale která svojí významností stojí alespoň za vyjmenování v této práci. Touto modifikací je 1-D dělení s ovlivnitelným pořadím přidělení submatic matice A procesům P_0 až P_{n-1}, čímž je docíleno rovnoměrnějších výpočetních časů jednotlivých procesů, což má za následek snížení času neefektivity T_O (Petersen, a další, 2004).

Existuje ještě celá řada dalších možných modifikací algoritmu běžného 1-D dělení, které však svým obsahem a složitostí implementace přesahují rozsah zkoumání této diplomové práce, proto je zde již uveden jen výpis názvů některých z nich:

- 1-D dělení s využitím pipeliningu při odesílání výsledků dělícího kroku,
- 2-D dělení,
- 2-D dělení s využitím pipeliningu,
- a další (Petersen, a další, 2004).

3.2.3 Čas paralelního běhu T_P

Pro pozdější výpočty a možnost porovnání teoretických předpokladů a naměřených dat je nezbytné odvodit vztah pro výpočet paralelního běhu implementace algoritmu T_P.

Tento čas je složen ze dvou složek: doba potřebná pro výpočty a doba strávená meziprocesorovou komunikací. První jmenovaná složka je vypočtena následujícím způsobem. Počet dělení použitých v k-té iteraci vnějšího cyklu algoritmu je možné vypočítat ze vztahu:

$$d = n - k - 1$$

kde d – je počet dělení,
n – je rozměr matice A,
k – je číslo iterace algoritmu.

Stejným vztahem jsou dány také počty násobení a eliminací v k-té iteraci pro každý proces P_i, kde platí:

$$k < i < n$$

kde i – je číslo procesu.

Pokud je brán v úvahu fakt, že jedna operace trvá jednu jednotku času, je možné čas strávený výpočty v k-té iteraci napsat jako:

$$T_{ck} = 3(n - k - 1)$$

kde T_{ck} – je čas strávený výpočty v k-té iteraci algoritmu.

Za předpokladu, že v okamžiku, kdy k-tý proces P_k provádí dělící krok, je ostatních $p - 1$ procesů nečinných, a za předpokladu, že ve chvíli kdy procesy P_{k+1} až P_{n-1} jsou zaměstnány prováděním eliminačních kroků, jsou procesy P_0 až P_k nečinné, je možné, jak uvádí (Ananth Grama, 2003), stanovit celkový čas strávený výpočty pro všechny iterace jako:

$$T_C = 3 \sum_{k=0}^{n-1} (n - k - 1) \approx \frac{3n(n-1)}{2}$$

kde T_C – je čas strávený výpočty pro všechny iterace algoritmu.

Druhá složka, kterou je čas potřebný pro meziprocesorovou komunikaci T_{com}, je podle (Ananth Grama, 2003) dána následujícím vztahem:

$$T_{com} = t_s \, n \log n + t_w \frac{n(n-1)}{2} \log n$$

kde T_{com} – je čas nezbytný pro meziprocesorovou komunikaci,
t_s – je čas nutný pro přijetí nebo odeslání jedné zprávy,
t_w – je čas potřebný pro přenesení jedné zprávy.

Výsledný vztah pro čas paralelního běhu T_P je potom možné vypočítat za pomocí tohoto vztahu:

$$T_P = T_C + T_{com}$$

$$T_P = \frac{3}{2} n(n-1) + t_s \, n \log n + \frac{1}{2} t_w n(n-1) \log n$$

Výsledný vztah je důkazem nižší časové náročnosti paralelní implementace v porovnání s implementací sériovou pro shodnou velikost n řešené matice A, protože již zmíněná veličina n zde dosahuje nejvyššího řádu n^2 na rozdíl od vztahu pro složitost sériové implementace, jež je odvozen v podkapitole 3.1.2, kde veličina n dosahuje nejvyššího řádu n^3.

3.2.4 Knihovna MPI

V průběhu implementace paralelní verze algoritmu Gaussovy eliminace byla použita knihovna MPI. Název této knihovny je zkrácením anglických slov Message Passing Interface, překlad tohoto názvu, jímž je „Rozhraní pro předávání zpráv“, napovídá její základní funkci, která je stručně popsána v následujících odstavcích.

Použitá open source varianta Open MPI je implementací MPI protokolu, jenž slouží k podpoře paralelních řešení výpočetních problémů při využití prostředí počítačových clusterů. Základním principem tohoto rozhraní je zasílání zpráv mezi prvky sítě, které může probíhat ve dvou variantách:

- přímé zaslání nebo příjem zprávy 1:1,
- zaslání nebo příjem jedné zprávy více uzly (mpi-forum.org, 2013).

Toto odesílání a příjem zpráv je pravděpodobně největší odlišností od kódu napsaného ve stejném jazyce pouze pro jeden procesor bez využití této knihovny. Kromě již zmíněného požadavku na komunikaci mezi procesy existuje ještě celá řada dalších, jež musí knihovna použitelná při paralelních řešeních v distribuované paměti splňovat. Stručný výpis dalších nejvýznamnějších vlastností Open MPI je uveden v následujícím výčtu:

- použití bariér pro synchronizaci činnosti procesů,
- značení procesů a možnost jejich jednoznačné identifikace,
- možnost identifikace a sledování zpráv,
- nemožnost použití sdílených proměnných mezi procesy,
- blokující nebo neblokující komunikační operace,
- použití standardních datových typů a možnost jejich použití při typové kontrole,
- komunikátory neboli skupiny procesů
- a mnoho dalších (Huss-Lederman, a další, 1998).

Při implementaci paralelní verze algoritmu Gaussovy eliminační metody byly použity především tyto: komunikátory, blokující / neblokující komunikace, speciální instrukce pro měření časů běhu, jež jsou podrobněji popsány v podkapitole 4.2, synchronizační bariéry a funkce pro základní manipulace s procesy a celou knihovnou.

Toto API rozhraní je ve své základní podstatě nezávislé na programovacím jazyce, protože se principiálně jedná o síťový protokol. V průběhu měření dat v experimentální části práce byla použita implementace této knihovny v „čistém“ jazyce C, i když existuje ještě celá řada dalších implementací ve vyšších programovacích jazycích, jež přinášejí větší komfort při samotné implementaci. Tento komfort při psaní kódu je ovšem za cenu nižší výkonnosti, což je v oblasti paralelních výpočtů zásadní kritérium (Gropp, a další, 1999). Použití implementace v „čistém“ jazyku C byl rovněž jeden z požadavků daných v podmínkách použití clusteru ve výzkumném centru NERSC, kde byla naměřena data pro

tuto diplomovou práci. Prostředí a podmínky jeho užití jsou podrobněji popsány v samostatné podkapitole 4.1.

3.3 Implementace zpětné substituce

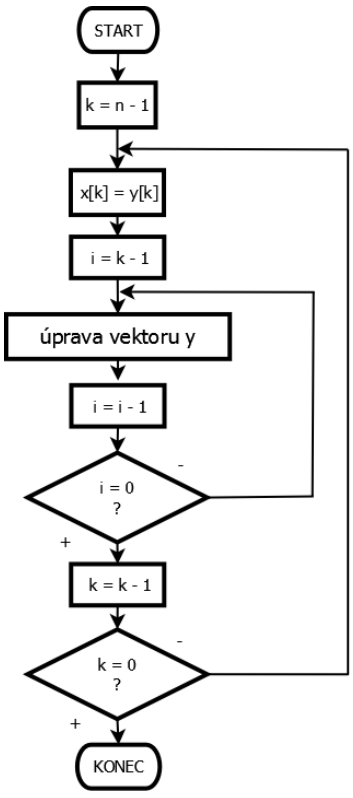
Po úpravě vstupní matice A a příslušných vektorů b a x je následujícím krokem implementace Gaussovy eliminační metody zpětná substituce. Tento výpočet je možné vysvětlit jako získání výsledných kořenů soustavy rovnic z upravené matice A (resp. U) a příslušných vektorů b a x (resp. y a x). Podrobněji jsou algoritmus a implementace zpětné substituce popsány v následujících odstavcích.

3.3.1 Sériová implementace zpětné substituce

V experimentální části byla použita jako druhý krok Gaussovy eliminace vždy sériová implementace zpětné substituce, protože naměřené časy běhů zkoumané v této diplomové práci se týkají pouze prvního kroku Gaussovy eliminační metody, jímž je úprava matice A a příslušných vektorů do formy, ze které lze vypočítat kořeny za použití popisované zpětné substituce.

Zpětná substituce byla použita vždy po ukončení měření až v průběhu závěrečného ověření správnosti vypočítaných hodnot. Tento postup je popsán v samostatné podkapitole 3.4.

Na následujícím obrázku je zobrazen základní sériový algoritmus zpětné substituce, který je principiálně založen na dvou vnořených cyklech.



Obrázek 13 - Sériová implementace zpětné substituce

Z uvedeného obrázku je patrné, že kořeny jsou vypočítávány v opačném pořadí, tj. od posledního $x[n-1]$ k prvnímu $x[0]$, takovým způsobem, že v každé iteraci vnějšího cyklu je vyřešen právě jeden kořen $x[k]$.

Princip sériového algoritmu zpětné substituce je založen na dosazení vypočteného kořenu do všech předchozích rovnic. Tento postup je proveden v rámci vnitřního cyklu dle tohoto předpisu:

$$y_n = y[i] - x[k] * U[i,k]$$

kde y_n – je vypočtená hodnota složky vektoru, která je následně zpětně uložena do $y[i]$,
$y[i]$ – je hodnota i-té složky vektoru y,
$x[k]$ – je kořen vypočtený v předchozí iteraci vnějšího cyklu,
$U[i,k]$ – je koeficient z matice U (resp. matice A).

Po skončení činnosti algoritmu jsou vypočtené kořeny uloženy ve vektoru x. Složitost této sériové implementace zpětné substituce je, jak uvádí (MERKLE, 2012), dána vztahem:

$$W \approx \frac{n^2}{2}$$

kde W – je složitost sériové implementace zpětné substituce,
n – je rozměr matice A (resp. U).

3.3.2 Paralelní implementace zpětné substituce

Zpětnou substituci je možné stejně jako první část Gaussovy eliminační metody paralelizovat. Existuje celá řada variant algoritmů, jež mohou být použity s určitým stupněm zrychlení běhu. V rámci této práce však nebudou podrobně vysvětleny, protože v experimentální části byla použita sériová implementace zpětné substituce jak při zkoumání sériové, tak při zkoumání paralelní implementace první části Gaussovy eliminační metody, kterou je úprava matice A na horní trojúhelníkovou matici U a upravení příslušných vektorů b a x.

Následuje výčet některých používaných metod paralelizace zpětné substituce:

- 1-D řádkové rozdělení matice A a příslušných vektorů b a x,
- 1-D rozdělení s využitím principu pipeliningu při odesílání výsledků ostatním procesům,
- 2-D řádkové a sloupcové rozdělení matice A a příslušných vektorů,
- 2-D rozdělení s využitím principu pipeliningu,
- kombinace výše uvedených s principem ovlivnitelného pořadí zpracování řádků[3],

[3] Použitím tohoto principu dojde k zmírnění vlivu problému s neaktivní oblastí matice A, jež se také vyskytuje při řešení zpětné substituce.

- a další.

3.4 Ověření správnosti implementací

Po implementaci sériové i paralelní verze Gaussovy eliminační metody bylo nezbytné provést jejich verifikaci. Použitý princip verifikace je možné popsat jako zpětné dosazení vypočtených kořenů do zdrojové soustavy rovnic a ověření platnosti rovnosti příslušných pravých a levých stran těchto rovnic. Z pohledu významnosti pro objekt zkoumání této práce se jedná spíše o doplněk, který byl použit hlavně ve fázích implementace a prvního testování, proto je v následujících odstavcích uveden jen základní princip.

Výsledkem paralelní i sériové implementace prvního kroku Gaussovy eliminační metody je upravená matice A a příslušné vektory. Použitím zpětné substituce, která je podrobněji popsána v předchozí podkapitole, na tato získaná data jsou vypočítány požadované kořeny, které jsou následně dosazeny do zdrojových rovnic. Tuto problematiku řeší algoritmus, jenž uvádí MERKLE (2012). Vstupem zmíněného algoritmu je horní trojúhelníková matice U, upravený vektor b, zjištěné kořeny a odchylka.

Důležitou poznámkou k implementaci ověřování rovnosti pravých a levých stran je vliv paměťové reprezentace desetinných čísel, zaokrouhlování a rozdílných datových typů na prosté porovnání dvou desetinných čísel v podmínce ve zdrojovém kódu. Tento problém řeší poslední zmíněný vstupní parametr, kterým je maximální možná odchylka. Skutečné ověření rovnosti poté neprobíhá pouhým porovnáním hodnot, ale dle následující podmínky:

$$|L - P| \leq \mathrm{d}$$

Kde L – je výsledná hodnota levé strany po dosazení kořenů,
P – je hodnota pravé strany,
d – je maximální přípustná hodnota odchylky.

Tento ověřovací algoritmus byl po verifikaci implementací zakomentován, aby nedocházelo ke zbytečnému prodloužení celkové doby jednoho experimentu při měřeném běhu na clusteru ve výzkumném centru NERSC, kde byl pro tato měření omezený čas, po který bylo možné používat potřebný počet nodů.

4 Porovnání sériové a paralelní implementace Gaussovy eliminace

4.1 Popis použitého prostředí v centru NERSC

V průběhu měření dat pro praktickou část této práce byly využívány nody prostředí s názvem Hopper ve výzkumném centru NERSC ve Spojených státech amerických, které bylo v době prováděných experimentů na 8. místě v celosvětovém hodnocení výkonu superpočítačů, který pravidelně uveřejňuje organizace Top500.org (2012).

4.1.1 Specifikace prostředí Hopper

V níže uvedené tabulce je seznam důležitých parametrů a specifikací prostředí Hopper ve výzkumném centru Národního vědeckého výpočetního centra pro výzkum v oblasti energie - NERSC dle oficiálních stránek (NERSC, 2013).

Tabulka 2 - Specifikace prostředí Hopper

Parametr / specifikace	**Hodnota**	**Jednotka**
Dostupný počet jader	153 408	-
Typ jader	Opteron 6172 12C, 2,1 GHz	-
Maximální výkon R_{max}	1 054.0	Tflop/s
Špička maximálního výkonu R_{peak}	1 288.6	Tflop/s
Příkon	2 910	kW
Architektura	Cray XE6	-
RAM	217	TB
Velikost disků	2	PB
Operační systém	Linux	-
Umístění	Berkley, USA	-
Web	http://nersc.gov	-
Účel	věda a výzkum	-

Z pohledu zaměření této práce jsou nejvýznamnějšími parametry maximální výkon R_{max} a špička maximální výkonu R_{peak}. První jmenovaný parametr je průměrný maximální počet proveditelných operací za jednu sekundu a druhý jmenovaný je teoreticky dosažitelný maximální počet proveditelných operací za jednu sekundu. Tyto hodnoty byly v praktické části práce použity pro výpočet časů běhu teoretických algoritmů.

Ostatní parametry a specifikace jsou spíše informativního charakteru. Například dostupný počet jader je pouze teoretický, ve skutečnosti se totiž počet použitých jader řídí prioritním mechanismem přidělování zdrojů, který je podrobněji popsán v následující podkapitole 4.1.2 spouštění úloh.

4.1.2 Spouštění úloh

Výpočetní úlohy jsou na prostředích ve výzkumném centru NERSC spouštěny za použití speciálního mechanismu prioritní fronty a sady instrukcí, jež jsou určeny pro obsluhu této fronty. Použitím takového mechanismu je zaručen relativně „spravedlivý“ přístup všech autorizovaných uživatelů dle jejich priority.

V následujících odstavcích jsou popsány procedury připojení a spuštění úloh, jež se týkají pouze prostředí Hopper, kde byla provedena měření dat pro praktickou část práce. V uvedených postupech nejsou uveřejněny všechny informace a data z důvodu dodržení přísných přístupových práv a podmínek užití, které musí každý uživatel NERSC prostředí potvrdit svým podpisem.

Protože prostředí Hopper je provozováno na operačním systému Linux, tak k přístupu postačí běžný linuxový terminál, IP adresa a získané uživatelské jméno a heslo. Připojení na server je poté realizováno za použití příkazu ssh, jehož syntaxe vypadá následovně:

```
ssh [parametry] user@IP-adresa
```

Po přihlášení uživatele je zobrazena zpráva s informacemi o dostupnosti jednotlivých prostředí. Dalším krokem je nakopírování souboru makefile[4] a všech zdrojových a řídících souborů projektu do lokální složky serveru. Jedním specifikem spuštění jakékoliv úlohy na tomto prostředí je použití speciálních řídících instrukcí s přesně definovanou jmennou konvencí a syntaxí – nedodržení tohoto pravidla a pokus o obejití prioritního mechanismu je trestáno okamžitým zamezením přístupu na všechny servery výzkumného centra NERSC.

V následující krátké ukázce zdrojového kódu jsou uvedeny řídící instrukce pro spuštění jednoho konkrétního experimentálního běhu paralelní implementace zkoumané Gaussovy eliminace.

```
#PBS -V
#PBS -q debug
#PBS -l mppwidth=200
#PBS -l mppnppn=3
#PBS -l mppdepth=1
#PBS -l walltime=0:10:00
#PBS -j oe

cd $PBS_O_WORKDIR
aprun -n 200 -N 3 ./gaussian 200 2000
```

V tabulce 3 uvedené níže jsou popsány a vysvětleny některé parametry z předchozího příkladu zdrojového kódu.

[4] Soubor makefile obsahuje pravidla a instrukce, jež jsou aplikovány v průběhu kompilace zdrojových kódů projektu.

Tabulka 3 - Parametry při spouštění úloh na serveru Hopper

Parametr / příkaz	Popis	Hodnota	Význam
-q	Typ fronty	debug	Slouží pro ladění kódu
-l walltime	Maximální čas běhu	0:10:00	Po 10 minutách běhu nastane timeout
-n	Počet spouštěných procesů	200	V průběhu výpočtu může být použito maximálně 200 procesů v jeden okamžik
-N	Rozložení procesů na nody	3	Maximálně 3 procesy poběží na jednom nodu
./gaussian 200 2000	Spuštění Gaussovy eliminace	200, 2000	Vstupní parametry: počet procesů, počet rovnic

Z pohledu zařazení do vybrané prioritní fronty čekajících úloh, kterou je v uvedeném příkladu debug fronta pro ladění kódu, jsou nejdůležitějšími parametry počet spouštěných procesů, rozložení procesů na nody a požadovaný maximální čas běhu. Na základě algoritmu, který ohodnotí požadavky a prioritu uživatele, je této úloze přiděleno pořadové číslo, na jehož základě je poté spuštěna. Je zřejmé, že úloha požadující alokaci velkého množství zdrojů nebo požadující velké časové okno serveru, získá větší pořadové číslo než úloha s menšími nároky nebo úloha od uživatele s vyšší prioritou.

Po dokončení běhu jsou všechny výstupy z aplikace uloženy na serveru ve složce uživatele ve speciálním souboru s následující jmennou konvencí:

```
název_ulohy.o-id_běhu
```

Část z jednoho konkrétního výstupu běhu aplikace je uvedena v následujícím příkladu:

```
Warning: no access to tty (Bad file descriptor).
Thus no job control in this shell.

I am 1; my runtime was: 0.014302
I am 2; my runtime was: 0.020257
I am 95; my runtime was: 0.213018
I am 93; my runtime was: 0.213063
…
Application 5091555 resources: utime ~133s, stime ~0s
 + ------------------------------------------------------------------------
 +        Job name: job-gaussian
 +          Job Id: 1147006.sdb
 +          System: hopper
 +     Queued Time: Sat Jan  7 10:50:05 2012
 +      Start Time: Sat Jan  7 10:53:18 2012
 + Completion Time: Sat Jan  7 10:53:24 2012
 +            User: charvat
 +        MOM Host: nid04218
 +           Queue: debug
 +  Req. Resources: other=QSUBPID:32710:hopper06,walltime=00:06:00
 +  Used Resources: cput=00:00:00,mem=0kb,vmem=0kb,walltime=00:00:06
 +     Acct String: mphpcd
 +   PBS_O_WORKDIR: /global/homes/c/charvat/implementation-05
 +     Submit Args: job-gaussian
 + ------------------------------------------------------------------------
```

V uvedeném příkladu je na prvních sedmi řádcích část výpisu měření časů běhu jednotlivých procesů. Ve zbytku úryvku jsou poté vypsána data s informacemi o celkovém běhu úlohy.

4.2 Metody měření časů

V průběhu provádění experimentů v rámci praktické části této práce bylo zásadní zvolit vhodnou metodu měření časů běhu aplikace a jednotlivých procesů. Pro tento účel byly použity speciální instrukce knihovny MPI, jež zaručují dostatečnou přesnost měření časových intervalů.

Takovou instrukcí je například MPI_Wtime(), která při zavolání vrací aktuální čas procesu, ve kterém běží. Příklad použití tohoto příkazu pro naměření časového intervalu je v úryvku kódu uvedeném níže.

```
//začátek měření
startTime = MPI_Wtime();

//měřený kód
…

//konec měření
endTime = MPI_Wtime();

//výpočet naměřeného času
time[k] = endTime – startTime;
```

Tato instrukce garantuje, že posloupnost vráceného času nebude v průběhu života procesu porušena. Návratová hodnota je typu double a jedná se o čas od začátku běhu procesu ve vteřinách s velkým počtem desetinných míst. Pro další použití je někdy nezbytné převést tento údaj na jiné vhodnější jednotky.

Měření časů za pomocí knihovny MPI může být provedeno ve dvou režimech:

- lokální časy procesů,
- globální synchronní čas.

Toto nastavení je řízeno globálním parametrem `MPI_WTIME_IS_GLOBAL`. V průběhu experimentů pro praktickou část této práce byla použita varianta lokální, protože nebylo nutné odečty časů běhu jednotlivých procesů mezi sebou nijak synchronizovat.

4.3 Porovnání teoretické a implementované sériové metody

4.3.1 Definice teoretické sériové metody

Před provedením srovnání teoretické a implementované sériové metody je nezbytné provést stanovení definičních podmínek běhu teoretické metody a následné vypočtení teoretických časů.

Sériový algoritmus běžící ve výzkumném centru NERSC na clusteru Hopper na jednom nodu má k dispozici maximální špičku výkonu $R_{peak} = 9{,}75\ Gflop/s$ (NERSC, 2013), což je maximální počet proveditelných operací na jednom jádru za jednu sekundu.

Tato hodnota však musí být pro potřeby výpočtu vynásobena koeficientem menším než jedna, protože ani ten nejlépe napsaný kód aplikace neběží s efektivitou sto procent. Pro potřeby zkoumání této práce byla zvolena hodnota 0,8, čímž je myšlena teoretická efektivita implementace sériového algoritmu 80 procent. Takto vysoké efektivity dosahují jen speciálně optimalizované implementace, které jsou většinou naprogramovány přesně na míru použitého prostředí nebo za podpory rychlých HW instrukcí. Příkladem takto optimalizované implementace může být nějaký algoritmus z knihovny, která využívá techniky BLAS pro výpočet násobení matic nebo vektorové počty.

Sériový čas běhu T_S je možné vypočítat dle následujícího vztahu:

$$T_S = \frac{W}{k * R_{peak}}$$

kde T_S – je sériový čas běhu algoritmu,
W – je složitost algoritmu,
R_{peak} – je maximální špičkový výkon,
k – je koeficient určující efektivitu implementace.

4.3.2 Porovnání teoretického algoritmu s implementovaným

Po dosazení vztahu pro výpočet složitosti sériové implementace $W = \frac{2}{3}n^3$, špičkového výkonu jednoho jádra prostředí Hopper, empiricky zvoleného koeficientu a rozměru matice například n = 4000 do rovnice uvedené výše dostaneme následující výsledek:

$$T_S = \frac{\frac{2}{3} * 4000^3}{0{,}8 * 9{,}75 * 10^9} \cong 5{,}47\ [\text{s}]$$

To znamená, že teoretický sériový algoritmus, který použije 80 procent možných zdrojů jednoho nodu, vyřeší Gaussovu eliminaci pro 4000 rovnic za 5,47 sekund.

V následující tabulce je uvedeno porovnání takovýmto způsobem vypočítaných teoretických a reálně naměřených hodnot.

Tabulka 4 - Porovnání teoretické a implementované sériové metody

n	naměřený Ts [s]	teoretický Ts [s]
1000	9.51651	0.085470085
2000	108.60862	0.683760684
4000	1077.42148	5.47008547
10000	-	85.47008547

Z uvedených hodnot v tabulce 4 je patrné, že pro velikost problému 10 000 nebyl naměřen čas běhu T_S, což bylo způsobeno překročením maximální možné doby spuštěné úlohy na serveru Hopper.

Teoretické časy a časy naměřené pro neoptimalizovanou obyčejnou sériovou metodu implementace Gaussovy eliminační metody se liší dokonce v několika řádech. To je způsobené velkým vlivem problému s hierarchií pamětí a neefektivnosti „obyčejné“ sériové implementace z tohoto pohledu. Tento jev je podrobněji popsán v samostatné podkapitole 1.6.6 Vliv hierarchie pamětí.

Míru efektivity (resp. neefektivity) sériové implementace je možné stanovit pro konkrétní naměřený čas běhu a pro konkrétní velikost problému zpětným dosazením hodnot do uvedené rovnice pro výpočet sériového času běhu T_S a její úpravou následovně:

$$T_S = \frac{W}{k * R_{peak}}$$

$$k = \frac{W}{T_S * R_{peak}}$$

Po dosazení složitosti $W = \frac{2}{3} * n^3$ pro sériovou implementaci, jež byla odvozena v podkapitole 3.1.2, získáme upravený vztah:

$$k = \frac{\frac{2}{3} * n^3}{T_S * R_{peak}}$$

Dosazením konkrétních hodnot například pro rozměr matice n = 4000 je možné vypočítat efektivitu využití HW prostředků takto:

$$k = \frac{\frac{2}{3} * 4000^3}{1077{,}4 * \ 9{,}75 * 10^9} = 4{,}06 * 10^{-3}$$

kde k – je koeficient určující efektivitu implementace.

Samotnou efektivitu využití hw prostředků v procentech E_h je možné vypočítat následovně:

$$E_h = 100 * k$$

$$E_h = 100 * \ 4{,}06 * 10^{-3} = 0{,}406\ \%$$

Výsledek vypočtené efektivity využití hw prostředků pouhé 0,4 % se může zdát na první pohled velice špatným v porovnání s 80 %, které byly použity pro výpočet teoretického „konkurenta“ sériové implementace.

Při bližším prozkoumání z pohledu problému s hierarchií pamětí, který má na tento výsledek největší vliv, je jasné, že teoretický optimalizovaný algoritmus s efektivitou 80 % by musel velice efektivně pracovat převážnou část času běhu na co nejnižších úrovních pamětí, které jsou velice rychlé. Přehled řádů velikostí a rychlostí paměti byl uveden v tabulce 1. Implementace takového kódu by však byla velice náročná a s velkou pravděpodobností závislá na konkrétním použitém HW testovacího prostředí, protože by vyžadovala použití speciálních HW instrukcí a jiných specifických optimalizačních technik.

Sériový algoritmus v průběhu své činnosti pracuje s maticí o velikosti n×n a s dvěma vektory o velikostech 1×n, jež obsahují data typu double. Jedna proměnná tohoto datového typu zabírá v jazyce C v paměti 8 bajtů. Z uvedených údajů je možné jednoduše vypočítat paměťovou náročnost neoptimalizovaného algoritmu. Například pro velikost problému n = 4000 je tento výpočet následující:

$$Použitá\ paměť = (n^2 + 2n) * 8$$

$$Použitá\ paměť = (4000^2 + 2 * 4000) * 8 = 128\ MB$$

Výsledná potřebná kapacita paměti není na první pohled pro problém velikosti n = 4000 nijak velká. Z pohledu přesunů a možnosti obsazení co nejnižších úrovní paměťové hierarchie už však tato hodnota významnou je, protože takto velké množství dat je v průběhu činnosti algoritmu rozprostřeno v prvních čtyřech úrovních paměťové hierarchie. Dle uvedených řádů časů přístupu v tabulce 1 je jasné, že přesuny mezi jednotlivými úrovněmi pamětí zaberou značnou část času běhu algoritmu, čímž výrazně přispějí ke snížení efektivity využití HW prostředků (především procesoru).

Existuje celá řada optimalizačních technik, jejichž cílem je ve většině případů snížení počtu přesunů mezi paměťovými úrovněmi a dosažení co největší hodnoty cache hit, která popisuje míru efektivního použití rychlé cache paměti. Tato problematika, jež byla částečně popsána v podkapitole 1.6.6, však svým obsahem značně přesahuje rámec zkoumání této diplomové práce a nebude zde dále rozváděna. Uvedené výpočty jsou pouze zjednodušeným vysvětlením relativně nízké hodnoty efektivity sériové implementace a spíše motivací k použití paralelní verze implementace Gaussovy eliminační metody.

4.4 Porovnání implementované sériové a implementované paralelní metody

V následujících podkapitolách je provedeno porovnání sériové a paralelní implementované metody Gaussovy eliminace z různých pohledů a za použití různých hodnotících kritérií.

4.4.1 Celkový čas běhu

Celkový čas běhu byl zvolen základní sledovanou veličinou při hodnocení implementovaných algoritmů. V následující tabulce jsou uvedeny celkové časy běhů implementací ve vteřinách takovým způsobem, že v každém sloupci tabulky jsou výsledky měření pro konstantní počet procesů p a měnící se velikost problému n a v každém řádku jsou naměřené časy pro konstantní velikost problému n a měnící se počet procesů p.

Tabulka 5 - Celkový čas běhu implementací v sekundách

	p = 1	**p = 4**	**p = 100**	**p = 500**	**p = 1000**
n = 1000	9.51651	0.61771	0.3364	3.49949	8.14885
n = 2000	108.60862	17.67346	0.84848	4.47707	14.48872
n = 4000	1077.4215	178.5312	4.26122	11.26539	26.22172
n = 10000	-	-	67.02792	67.21094	121.3873

Z uvedené tabulky je na první pohled patrné, že pro největší velikost problému n = 10000 nebyla naměřena hodnota celkového času běhu pro jeden a čtyři procesy. To je způsobeno tím, že v průběhu experimentu došlo k překročení maximálního možného času běhu úlohy. Tento limit je nastaven na prostředí Hopper jako prevence vytěžování nodů „zaseknutými“ nečinnými úlohami.

Dále je možné z uvedených výsledků měření vypozorovat, že všechny časy pro paralelní verzi implementace algoritmu, tzn. používající více než jeden proces, jsou nižší než pro verzi sériovou, která může použít pouze jeden proces.

Už při takto jednoduchém porovnání se však projevil jeden významný jev, který je pro paralelní úlohy typický. Je jím postupný pokles celkového času běhu s rostoucím počtem procesů pro konstantní velikost problému, který se však od určitého množství procesů p zastaví a následně zcela otočí v nárůst. Tato tendence, jež je prokázána ještě v průběhu vyhodnocení dalších měřených nebo vypočítaných veličin v následujících podkapitolách, je dána nárůstem nezbytné komunikace a synchronizace pro narůstající počet procesů, čímž dojde ke snížení procentuálního poměru času věnovaného výpočtům a tím i ke snížení efektivity. Tento jev je možné pozorovat například pro velikost řešeného problému n = 4000, kdy pro počet procesů p = 100 je celkový čas běhu nejmenší a s přibývajícím počtem procesů dochází k jeho nárůstu.

4.4.2 Čas věnovaný pouze výpočtům

Další sledovanou veličinou byl čas, který daná implementace strávila pouze výpočetními úlohami. To znamená, že se jedná o čas běhu implementace, který v sobě nezahrnuje dobu nutnou pro komunikaci a synchronizaci procesů. Výsledky tohoto měření jasně korespondují s jevem popsaným v předchozí podkapitole a dokonce jej potvrzují.

V následující tabulce je zobrazena procentuální část celkového času běhu, jež byla věnována pouze výpočtům v průběhu jednotlivých měření. Struktura tabulky a jejich popisků je obdobná jako pro tabulku 5, kde jsou ve sloupcích vypsána data pro konstantní počet procesů p a kde jsou v řádcích vypsána data pro jednu velikost problému n.

Tabulka 6 - Čas věnovaný pouze výpočtům - % z celkového času běhu

	p = 1	**p = 4**	**p = 100**	**p = 500**	**p = 1000**
n = 1000	100	95.52266	5.156064	0.079955	0.028078
n = 2000	100	99.65961	24.96287	0.472273	0.080235
n = 4000	100	99.13235	52.79981	4.397806	0.372607
n = 10000	-	-	80.30173	11.99591	4.345358

Procentuální hodnoty času věnovaného výpočtům v tabulce 6 byly vypočítány dle tohoto postupu:

$$T_C[\%] = \frac{T_C}{T_{all}} * 100$$

kde T_C – je čas strávený výpočty v sekundách,
T_{all} – je celkový čas běhu,
T_C [%] – je čas strávený výpočty v procentech z celkového času běhu.

Čas věnovaný výpočtům v sekundách T_C musel být naměřen zvlášť při samostatném opětovném běhu experimentu se zakomentovanými částmi kódu pro měření celkového času běhu algoritmu takovým způsobem, aby nedocházelo k ovlivnění naměřených hodnot.

V uvedené tabulce je hned na první pohled patrné, že hodnoty pro sériovou implementaci s jedním procesem (p = 1) jsou rovny 100 %. To však znamená pouze prostý fakt nepotřebnosti komunikace a synchronizace v případě použití jednoho procesu pro spuštění sériové implementace Gaussovy eliminační metody, čímž nedochází k žádné režii navíc a všechen čas je věnován pouze výpočtům, což může vypadat zajímavě, avšak hned na řádku pro rozměr n = 10000 není pro počet procesů p = 1 a p = 4 uvedena žádná hodnota, protože nebylo možné časy běhu vůbec naměřit ve stanoveném časovém limitu.

V ostatních sloupcích tabulky je možné vypozorovat tendenci, jež byla popsána v předchozí podkapitole. Například pro velikost řešeného problému n = 4000 je ve sloupcích pro p = 1 a pro p = 4 poměrně vysoká procentuální hodnota, jejíž příčina již

byla vysvětlena v předchozím odstavci. V následujícím sloupci pro 100 procesů je poměr času věnovaného výpočtům přes 52 procent, což je sice hodnota nižší než pro předchozí sloupce, ale z pohledu celkového času běhu je tato kombinace velikosti problému a počtu procesů lepší. S rostoucím počtem použitých procesů však dojde k výraznému poklesu času věnovaného výpočtům, jako je tomu pro p = 500 a pro p = 1000, protože je již převážná část času běhu věnována komunikaci, synchronizaci a čekání, což výrazně snižuje efektivitu použití HW prostředků a značně prodlužuje celkový čas běhu.

4.4.3 Zrychlení

Další srovnávanou veličinou je zrychlení, jež bylo podrobněji popsáno v podkapitole 1.6.2. V následující tabulce, která má podobnou strukturu rozložení popisků řádků a sloupců jako tabulky 5 a 6, jsou uvedeny vypočítané hodnoty zrychlení, jež jsou vztaženy vzhledem k rychlosti sériové implementace. Teoreticky je totiž možné uvést normované srovnání, které je vztaženo k nějaké referenční hodnotě zrychlení. Z pohledu vypracování této práce je však významnější uvést přímé porovnání sériové a paralelní implementace.

Tabulka 7 - Srovnání zrychlení sériové a paralelní implementace

	p = 1	**p = 4**	**p = 100**	**p = 500**	**p = 1000**
n = 1000	1	15.40611	28.28927	2.719399	1.167835
n = 2000	1	6.145295	128.0037	24.25886	7.496081
n = 4000	1	6.034922	252.8434	95.63996	41.08889
n = 10000	-	-	-	-	-

Tato veličina nemá uvedenou žádnou jednotku, protože se jedná o poměrové vyjádření srovnání rychlosti sériové a paralelní implementace. Zjednodušeně by se dalo nenormované zrychlení popsat jako vyjádření, kolikrát je paralelní implementace rychlejší než implementace sériová.

Vzhledem k definici zrychlení jsou v prvním sloupci pro počet procesů p = 1 uvedeny samé jedničky, protože sériová implementace je stejně rychlá v porovnání sama se sebou. V posledním řádku nejsou oproti tomu uvedeny žádné vypočítané hodnoty, protože z důvodu překročení maximální možné doby experimentu nebylo možné naměřit čas běhu sériové implementace pro počet procesů p = 1, a tudíž nebylo možné ani vypočítat porovnání rychlosti tohoto běhu s rychlostmi paralelní implementace.

V ostatních sloupcích tabulky jsou uvedeny hodnoty, které jsou vždy větší než jedna, což znamená, že paralelní implementace je pro tyto kombinace počtu procesů p a velikosti řešeného problému n rychlejší než implementace sériová.

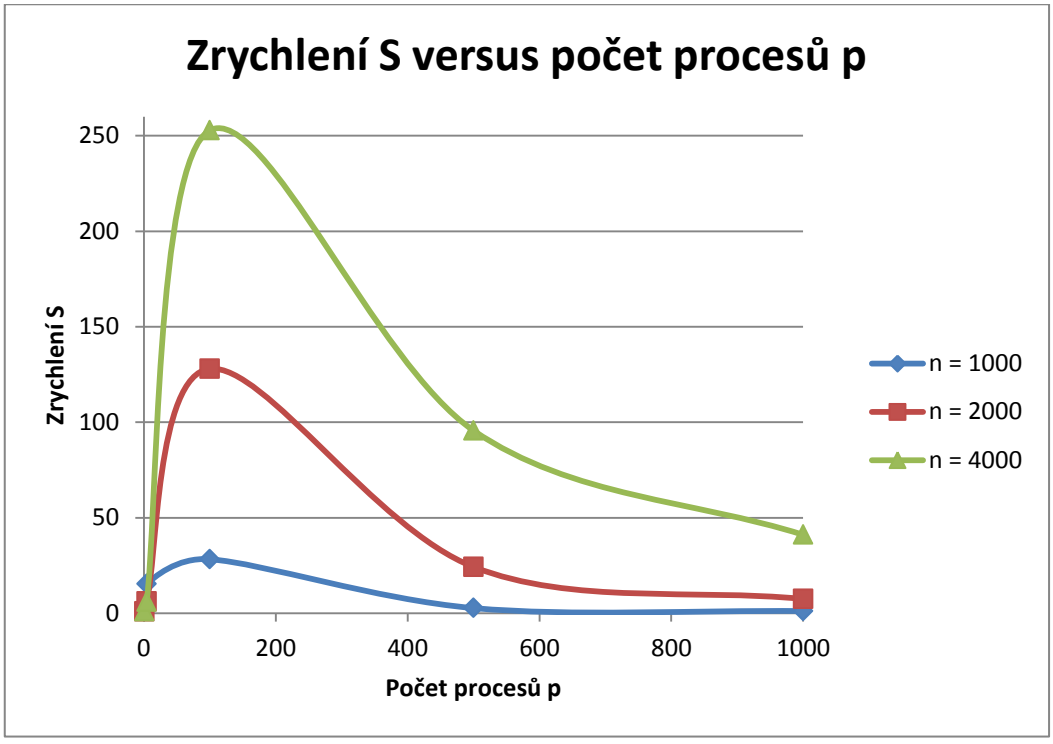
Na příkladu pro velikost n = 4000 a počet procesů p = 500 je vysvětlen jednoduchý způsob výpočtu zrychlení. Jedná se totiž o prostý poměr sériového a paralelního času běhu dle následujícího předpisu, který byl podrobněji popsán v podkapitole 1.6.2:

$$S = \frac{T_S}{T_p}$$

$$S = \frac{1077{,}421}{95{,}64} \cong 95{,}6$$

Teoreticky by neměla hodnota zrychlení překročit počet použitých procesů, jak uvádí (MERKLE, 2012), v praxi však tato situace nastat může. Příkladem je hodnota pro n = 2000 a p = 100, kde je vypočítané zrychlení rovno přibližně 128. Toto je dáno velikou sekvenčností sériového algoritmu a jeho implementací za použití tří vnořených cyklů se složitostí obsahující třetí mocninu n, což může způsobit, že skutečný čas běhu bude pro určitou velikost řešeného problému n vlivem hierarchie pamětí a nemožnosti efektivního použití času procesoru pro výpočty výrazně delší, než by byla teoreticky vypočtená hodnota.

Při detailnějším prozkoumání hodnot v jednotlivých řádcích je opět patrná tendence jejich nárůstu do určitého počtu procesů p, která je poté vystřídána jejich poklesem. Názorněji je tento jev viditelný v následujícím obrázku 14, kde je uveden graf závislosti zrychlení na počtu použitých procesů p pro dané velikosti řešeného problému n.



Obrázek 14 - Zrychlení S versus počet procesů p

Tato tendence grafu není obecným pravidlem pro všechna paralelní řešení úloh, ale je velice častá, jak bylo podrobněji popsáno v podkapitole 1.6.5 na obrázku 7. Ze zobrazených křivek pro jednotlivé velikosti problému je možné rozpoznat ještě jeden jev, kterým je nárůst zrychlení s nárůstem již zmíněné velikosti problému, což se dá logicky vysvětlit menší ztrátou času při komunikaci, a s větší velikostí problému logicky souvisejícím delším časem, který byl stráven výpočty, což je ve finále zdrojem benefitu ve formě většího zrychlení S.

4.4.4 Efektivita

Dalším hodnotícím kritériem při porovnání sériové a paralelní implementace Gaussovy eliminační metody byla zvolena efektivita, jež je zobrazena v následující tabulce číslo 8, která má stejnou strukturu a popisky jako tabulky předchozí.

Tabulka 8 - Porovnání efektivity sériové a paralelní implementace

	p = 1	**p = 4**	**p = 100**	**p = 500**	**p = 1000**
n = 1000	1	3.851528	0.282893	0.005439	0.001168
n = 2000	1	1.536324	1.280037	0.048518	0.007496
n = 4000	1	1.50873	2.528434	0.19128	0.041089
n = 10000	-	-	-	-	-

Nejprve je nezbytné uvést, že se nejedná o „klasickou“ efektivitu, jež nabývá hodnot v rozsahu od nuly do jedné, ale že se jedná o poměrové vyjádření, kolikrát je daná paralelní implementace efektivnější než implementace sériová. Tato zjednodušená definice efektivity vysvětluje samé jedničky ve sloupci pro počet procesů p = 1, protože při porovnání sériové implementace se sebou samou dostaneme právě tento výsledek.

Názorněji je možné definici efektivity vysvětlit na konkrétním příkladu jejího výpočtu pro velikost řešeného problému n = 4000 a počet procesů p = 100, který vypadá následovně:

$$E = \frac{T_S}{T_p * p}$$

$$E = \frac{1077{,}42}{4{,}26 * 100} \cong 2{,}53$$

Což znamená, že pro tuto velikost problému n = 4000 a počet procesů p = 100 dosahuje paralelní implementace 2,53 krát větší hodnoty efektivity.

Tuto hodnotu je možné vyčíslit za použití odvození a vypočtení efektivity využití HW E_h sériové implementace v procentech v podkapitole 4.3.2, kde byla její hodnota vypočtena jako 0,406 procent. Jednoduchým vynásobením dostaneme následující:

$$E_{hp} = E * E_h$$

$$E_{hp} = 2{,}53 * 0{,}406 \cong 1{,}03\ \%$$

kde E_{hp} – je vypočtená hodnota efektivity paralelní implementace pro konkrétní velikost problému n a počet procesů p v procentech,

E – je poměr efektivity sériové a paralelní implementace pro konkrétní velikost problému n a počet procesů p,
E_h – je efektivita sériové implementace v procentech.

Vypočtená hodnota, jež se na první pohled zdá velice nízká, je několikanásobně větší než efektivita sériové implementace.

Při dalším prozkoumání tabulky je patrné, že pro velikost řešeného problému n = 10000 nejsou uvedeny žádné hodnoty poměru efektivity, což je způsobeno nemožností naměření času běhu sériové implementace T_S pro tuto velikost n, který je nezbytný pro pozdější výpočet efektivity.

Na uvedených datech je možné také potvrdit tendenci nárůstu a následného poklesu zrychlení S, která byla podrobně popsána v předchozí podkapitole 4.4.3. Například pro již několikrát zmiňovanou velikost problému n = 4000 je tento jev patrný na první pohled. Pro počet procesů p = 100 je hodnota poměru efektivity rovna přibližně 2,53. S nárůstem počtu procesů na p = 500 však dojde ke skokovému snížení až o jeden řád na přibližně 0,2. Toto výrazné snížení, které je možné pozorovat na více místech v uvedené tabulce, poté koresponduje s již popisovaným poklesem zrychlení S v tabulce 7, s poklesem času věnovaného výpočtům T_C v tabulce 6 a se začátkem nárůstu celkového paralelního času běhu v tabulce 5.

Dalším jevem, který se vyskytuje v uvedené tabulce číslo 8 především pro větší počty použitých procesů p, je výskyt hodnot, jež jsou nižší než jedna, což prakticky znamená menší efektivitu paralelní implementace pro tento počet procesů v porovnání s efektivitou sériové implementace pro jeden proces. Nyní je však nezbytné podotknout, že celkový čas paralelního běhu je pro tyto kombinace počtu procesů p a velikosti řešených problému n stále nižší než čas sériového běhu pro tyto velikosti problému n, což je v oblasti řešení výpočetních úloh velice významná vlastnost.

Na druhou stranu existuje extrémní případ, kdy použitím přílišného počtu procesů p a následným nadměrným rozdělením problému dojde vlivem výrazné neefektivity k dosažení dokonce horšího času běhu pro paralelní implementaci v porovnání s implementací sériovou.

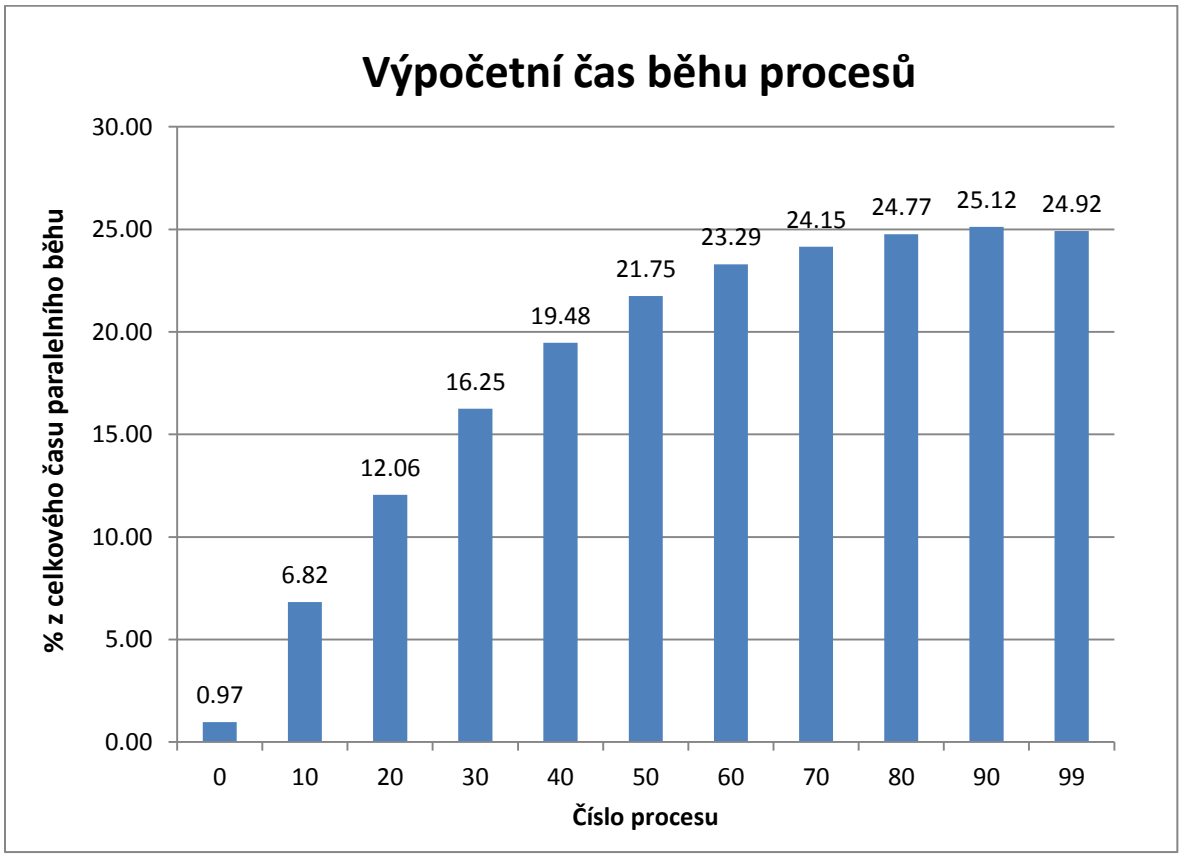
Dalším extrémním případem, který je celkem běžný pro velké hodnoty velikosti řešeného problému n, je nemožnost dokončení výpočtu ve stanoveném časovém limitu nebo dokonce absolutní nemožnost dokončení výpočtů. Důvodem k tomu je ve většině případů použití nízkého počtu procesů p vzhledem k velikosti problému n.

Stanovení optimální hodnoty p pro řešenou velikost problému n se poté stává výsledkem dvou protichůdných požadavků na cílové řešení. Prvním z nich je dosažení co nejnižšího času běhu a druhým je co nejefektivnější využití dostupných HW prostředků za účelem snížení ceny výsledného řešení. Tato úvaha je vždy závislá na konkrétním případu paralelizované úlohy, protože cena a čas dokončení přijatelný pro jeden problém nemusí být přijatelný pro jiný.

4.4.5 Výpočetní časy jednotlivých procesů

Bylo provedeno několik speciálních měření za účelem bližšího prozkoumání zdrojů neefektivity zvolené varianty paralelní implementace Gaussovy eliminační metody, jejichž sledovanou veličinou byl čas běhu věnovaný pouze výpočtům jednotlivých procesů.

Výsledek tohoto experimentu pro velikost problému n = 2000 a počet procesů p = 100 je zobrazen na následujícím obrázku:



Obrázek 15 - Výpočetní čas běhu jednotlivých procesů

Na obrázku 15 je uveden graf, jenž zobrazuje procentuální část z celkového času paralelního běhu, kterou věnoval daný proces pouze výpočtům. Tendence hodnot grafu, jež je na první pohled naprosto patrná, je charakteristickým jevem zvolené varianty paralelizace Gaussovy eliminační metody pomocí 1-D dělení a byla již částečně popsána

v podkapitole 3.2 v odstavcích o problému s neaktivní oblastí. V přílohách A, B a C jsou uvedeny grafy pro další kombinace počtů procesů p a velikost problému n = 2000.

Největší hodnota procentuální části z celkového času věnované pouze výpočtům je pouhých 25 procent, což prakticky znamená, že i ten nejaktivnější proces stráví převážnou většinu času svého běhu čekáním nebo komunikací. V případě pro tuto kombinaci velikosti n a počtu procesů p je to dokonce 75 procent času. Tento jev se zákonitě musí „podepsat“ na nízké hodnotě výsledné efektivity.

Dalším zjištěním tohoto měření je potvrzení faktu vysoké nerovnoměrnosti vytížení jednotlivých procesů, jež je dána sekvenčností Gaussovy eliminační metody. Nejvíce patrný je tento jev při porovnání výpočetního vytížení nultého a posledního procesu, kde je rozdíl těchto hodnot největší. Toto negativum použití 1-D dělení je další příčinou nižších hodnot efektivity paralelní implementace.

Existuje celá řada řešení výše popsaných problémů. Jedním z nich je ovlivnění přidělení jednotlivých submatic procesům, jež bylo popsáno v podkapitole 3.2.2, čímž dojde k rovnoměrnějšímu vytížení procesů a ke zvýšení efektivity.

Další možností, která může částečně ovlivnit nerovnoměrnost vytížení jednotlivých procesů a tím zlepšit efektivitu, je použití techniky 2-D dělení společně s technikou pipeliningu volání prováděných komunikačních operací. Tato technika je však mnohem náročnější na implementaci než použité 1-D dělení, toto hledisko je podrobněji popsáno v následující podkapitole 4.4.6. Jak uvádí (Ananth Grama, 2003) jedná se o výhodnější verzi implementace z pohledu efektivity.

4.4.6 Náročnost implementace vs. velikost problému n

Posledním ne méně významným hlediskem, jež bylo bráno v úvahu při porovnávání sériové a paralelní implementace, je samotná náročnost implementace zvolené metody. Ač se může toto hledisko zdát na první pohled nevýznamné, je nutné jej vzít v úvahu především pro menší velikosti řešeného problému n, protože pro větší velikosti n bylo v předchozích podkapitolách zjištěno, že je použití sériové implementace ve většině případů časově nevýhodné nebo dokonce nemožné.

Existují samozřejmě specifické případy i pro řešení úloh větších rozsahů, kdy může dojít díky použití nadměrného množství procesů p k časové nevýhodnosti, jak již bylo podrobněji popsáno na konci podkapitoly 4.4.4, jež tuto problematiku zkoumá z pohledu efektivity.

Oproti tomu pro malé velikosti řešených úloh n vyvstává nová otázka, zda je vůbec výhodné se zabývat i pro malý problém poměrně složitou paralelizací a implementací paralelní verze algoritmu. Toto hledisko je vždy závislé na konkrétním případu a konkrétních požadavcích. Pro zkoumanou problematiku Gaussovy eliminace je možné poměrně jednoduchým výpočtem za použití již odvozených vztahů a vypočtených hodnot v předchozích kapitolách vypočítat předpokládaný čas běhu sériové implementace

pro konkrétní velikost problému n na konkrétním HW. Například pro zvolenou velikost n = 100 a stejné HW prostředky, jenž byly použity v průběhu experimentů praktické části této práce, je výpočet následující:

$$T_S = \frac{W}{k * R_{peak}} = \frac{\frac{2}{3}n^3}{k * R_{peak}}$$

$$T_S = \frac{\frac{2}{3} * 100^3}{4{,}06 * 10^{-3} * 9{,}75 * 10^9} = 0{,}01684\ [s]$$

kde T_S – je sériový čas běhu implementace,
W – je složitost problému,
n – je velikost řešené úlohy,
R_{peak} – je maximální možný počet operací jednoho jádra za sekundu,
k – je vypočtená hodnota konstanty určující efektivitu pro velikost problému n = 4000.

Hned po dokončení tohoto výpočtu je nutné uvést, že se jedná o pouhý odhad sériového času běhu T_S pro velikost řešeného problému n = 100, který je dán předpokladem použití stejného hardwaru a stejné konstanty k určující efektivitu, jež byla vypočtena v podkapitole 4.3.2 pro velikost problému n = 4000. Teoreticky je tedy možné předpokládat, že tato hodnota konstanty k bude pro menší velikost problému n vyšší, protože sériová implementace pravděpodobně nenarazí na tak veliký problém s hierarchií pamětí pro takto malou velikost řešeného problému, jako tomu bylo v případě n = 4000.

Na základě tohoto jednoduchého výpočtu je možné provést odhad sériového času běhu T_S a následné vyhodnocení nutnosti použití paralelní implementace pro konkrétní velikost úlohy, jež je většinou dána požadavky na přípustný maximální čas dokončení této úlohy.

4.5 Porovnání teoretické sériové a implementované paralelní metody

Porovnání teoretické sériové a implementované paralelní metody je uvedeno až na konci této diplomové práce jen jako doplněk, protože hlavním cílem práce bylo porovnání konkrétních implementovaných metod. Výsledek tohoto srovnání však přinesl několik zajímavých výsledků a nové hledisko, jež může být použito v závěrečném zhodnocení porovnání implementací a možností jejich vylepšení.

4.5.1 Časy běhu

Základní parametrem pro srovnání teoretické sériové a paralelní implementované metody je celkový čas běhu, který je uveden v následující tabulce, jež má podobnou strukturu a popisky jako předchozí tabulky obdobných srovnání:

Tabulka 9 - Porovnání časů běhu teoret. sériové a paralelní implementované metody

	p = 1 (teoret.)	p = 4	p = 100	p = 500	p = 1000
n = 1000	0.085470085	0.61771	0.3364	3.49949	8.14885
n = 2000	0.683760684	17.67346	0.84848	4.47707	14.48872
n = 4000	5.47008547	178.5312	4.26122	11.26539	26.22172
n = 10000	85.47008547	-	67.02792	67.21094	121.3873

V uvedených datech tohoto zajímavého srovnání teoretických hodnot v prvním sloupci a naměřených hodnot v ostatních sloupcích je možné vypozorovat dvě zásadní informace.

První z nich je, že pro velikosti n = 1000 a n = 2000 dosahuje teoretický sériový algoritmus menších časů běhu než implementovaný paralelní, což může na první pohled vypadat jako ne příliš dobrý výsledek pro paralelní verzi. Avšak z pohledu efektivity teoretické sériové varianty, která je v tomto případě zvolena 80 procent, což je mnohonásobně vyšší než skutečná hodnota implementované sériové varianty, je tento výsledek teoreticky skutečně možný, protože takto vysoká efektivita odpovídá velice optimalizovanému kódu napsanému přímo na míru použitého hardware.

Druhou zásadní informací, kterou je možné nalézt v datech uvedené tabulky, je výskyt nižších časů běhu paralelní implementované varianty pro velikosti n = 4000 a n = 10000, což je pozitivní výsledek srovnání ve prospěch paralelního řešení problému vzhledem ke zvolené velké efektivitě sériové implementace.

Tato dvě zjištění potvrzují úvahu uvedenou v podkapitole 4.4.6 o vlivu velikosti problému n na vhodnost použití paralelní implementace pro řešení této úlohy, protože dokazují, že je teoreticky možné optimalizací sériové implementace dosáhnout kratších časů běhu pro malé velikosti problému n a naopak pro větší velikosti problému n dokazují fakt, že ani použitím vysoce optimalizovaného sériové algoritmu nelze dosáhnout lepších časů běhu pro takto velké úlohy.

4.5.2 Zrychlení

Dalším kritériem porovnání je, stejně jako tomu bylo při porovnání implementovaných algoritmů, zrychlení S, jež je uvedeno v následující tabulce:

Tabulka 10 - Porovnání zrychlení teoret. sériové a paralelní implementované metody

	p = 1 (teoret.)	p = 4	p = 100	p = 500	p = 1000
n = 1000	1	0.138366	0.254073	0.024424	0.010489
n = 2000	1	0.038689	0.805865	0.152725	0.047193
n = 4000	1	0.030639	1.28369	0.485566	0.208609
n = 10000	1	-	1.275142	1.271669	0.704111

Zrychlení S je zde vypsáno a vypočteno obdobným způsobem a za použití stejného vztahu jako při porovnání v podkapitole 4.4.3 včetně příčiny výskytu jedniček v prvním sloupci a nevypočtené hodnoty v druhém sloupci.

Data v tabulce menší než jedna potvrzují fakt výhodnosti použití teoretické sériové implementace pro malé velikosti problému n, jež byla popsána v předchozí podkapitole 4.5.1, protože hodnota zrychlení S menší než jedna značí menší rychlost paralelního algoritmu.

Oproti tomu hodnoty v tabulce, jež jsou větší než jedna, potvrzují druhý fakt, který byl rovněž podrobněji vysvětlen v podkapitole 4.5.1, jímž je výhodnost použití paralelní verze implementace pro větší velikosti problému n, protože hodnota zrychlení S větší než jedna značí větší rychlost paralelní implementace algoritmu.

4.5.3 Efektivita

Uvedení srovnání z pohledu efektivity má v tomto případě pouze informativní charakter, protože pro teoretický sériový algoritmus byla zvolena efektivita 80 procent, což je velmi vysoká hodnota v porovnání s efektivitou použité sériové implementace, která je přibližně 0,4 procenta, což má za následek velmi nízké hodnoty poměru porovnání teoretické sériové a implementované paralelní metody, jež je uvedeno v následující tabulce:

Tabulka 11 - Porovnání efektivity teoret. sériové a paralelní implementované metody

	p = 1 (teoret.)	**p = 4**	**p = 100**	**p = 500**	**p = 1000**
n = 1000	1	0.034592	0.002541	4.88E-05	1.05E-05
n = 2000	1	0.009672	0.008059	0.000305	4.72E-05
n = 4000	1	0.00766	0.012837	0.000971	0.000209
n = 10000	1	-	0.012751	0.002543	0.000704

Hodnoty poměru efektivit v tabulce 11 jsou vypočteny za použití obdobných vztahů jako v tabulce 8 s tím rozdílem, že pro výpočet byly použity teoretické časy běhu sériové implementace T_S z tabulky 4.

Výsledky tohoto porovnání jsou takovým způsobem ovlivněny hodnotou zvolené teoretické efektivity sériové implementace, že jediná informace, kterou je možné z těchto dat získat, je potvrzení tendence postupného nárůstu a následného poklesu efektivity při nárůstu počtu použitých procesů p pro konstantní velikost řešené úlohy n stejně, jako je tento jev popsán v podkapitole 4.4.4, kde je porovnána efektivita implementovaných algoritmů.

Závěr

Cílem teoretické části práce bylo provedení úvodu do problematiky paralelního programování výpočetních úloh, jejich případných optimalizací a vysvětlení nezbytných základních pojmů. Dalším cílem této části bylo vysvětlení teoretického postupu paralelizace Gaussovy eliminační metody v distribuované paměti.

Výše uvedené teoretické cíle byly splněny v dostatečné míře v rámci prvních dvou kapitol.

Cílem praktické části této diplomové práce byla implementace vybrané metody paralelizace Gaussovy eliminační metody, její otestování na serveru Hopper ve výzkumném centru NERSC a porovnání získaných dat s hodnotami pro sériovou neparalelní metodu.

V rámci praktické části byla implementována kompletně funkční paralelizace Gaussovy eliminace v distribuované paměti za použití techniky 1-D rozdělení matice A a vstupních vektorů b a x, jež pro svou činnost používala funkce z knihovny MPI. Rovněž byla implementována sériová neparalelní verze algoritmu, jejíž časy běhu byly následně porovnány s již zmíněnou paralelní verzí algoritmu.

Součástí implementace bylo také naprogramování jednoduchého „frameworku“, který byl nezbytný pro běh experimentů, měření časů, vyhodnocení dat a ověření správnosti výsledků práce. Pro ověření správnosti vypočtených výsledků jednotlivých běhů bylo nutné implementovat rovněž zpětnou substituci včetně dosazení vypočtených kořenů zpět do rovnic a ověření rovnosti pravých a levých stran v požadované přesnosti.

Všechny zdrojové kódy aplikace, jež jsou přiloženy na CD k této práci, byly napsány v programovacím jazyce C, který je díky své vysoké efektivitě standardem ve výzkumném centru NERSC.

Práce na druhé skupině cílů praktické části, mezi něž patřilo měření hodnotících veličin a jejich následné zpracování a porovnání, přineslo závěry, které byly očekávány v teoretických předpokladech.

Nejzásadnějším závěrem je zjištění, že pro menší velikosti řešeného problému nemusí být vždy vhodné použít paralelní řešení implementace a pro větší velikosti problémů může být použití paralelní verze implementace dokonce nezbytné. Tyto dvě tendence byly dokázány a vysvětleny na naměřených datech. První je způsobena přílišnou ztrátou času běhu při nadměrné meziprocesorové komunikaci vzhledem k velikosti problému a druhá je způsobena nemožností vyřešení úlohy větší velikosti za použití pouhé sériové implementace v požadovaném čase.

Dalším velmi zásadním závěrem práce je dokázání vlivu počtu použitých procesů na celkovou efektivitu implementace, kde s rostoucím počtem procesů dochází k nárůstu efektivity až do „okamžiku zlomu“, kdy přidání každého dalšího procesu může způsobit

značný pokles efektivity způsobený již zmíněnou nadměrnou meziprocesorovou komunikací, která může mít za následek dokonce prodloužení celkového času běhu.

Významným zjištěním, které přineslo zkoumání naměřených dat v praktické části, je také míra vlivu nárůstu počtu použitých procesů na rychlost růstu efektivity. V naměřených datech byla totiž zjištěna další tendence, která se projevovala ještě před dosažením již zmíněného „okamžiku zlomu“ a kterou bylo možné popsat jako zpomalení nárůstu efektivity E a s tím souvisejícího snížení nárůstu zrychlení S pro rostoucí počet procesů, což lze zjednodušeně vysvětlit tak, že od určitého množství použitých procesů je na zvážení, zda je z ekonomického pohledu vůbec výhodné použití většího množství procesů, které logicky souvisí s použitím většího množství HW. Tato úvaha je velmi závislá na konkrétním typu řešené úlohy a požadavcích na celkovou dobu běhu implementace.

Práce na tomto projektu pro mě byla velikým přínosem, protože již při tvorbě teoretické části jsem si ujasnil některé důležité pojmy z tohoto oboru a při programování a experimentech v praktické části jsem si v praxi vyzkoušel nastudované algoritmy implementací a obohatil jsem se o zkušenost s prací a testováním na serverech velkého výzkumného centra NERSC, jenž byly v té době na 8. místě ve světovém žebříčku superpočítačů.

Literatura

Almasi, G. S. a Gottlieb, A. 1994. *Highly Parallel Computing.* Redwood City : Benjamin/Cummings, 1994.

Ananth Grama, Anshul Gupta, George Karypis a Vipin Kumar. 2003. *Introduction to Parallel computing (2nd edition).* 2003. ISBN 978-0-201-64865-2.

Cormen, T. H., Leiserson, C. E. a Rivest, R. L. 1990. *Introduction to Algorithms.* New York : MIT Press, 1990.

Eager, D. L., Zahorjan, J. a D., Lazowska E. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers.* 1989.

Foster, Ian a Addison-Wesley. 1995. *Designing and Building Parallel Programs.* 1995.

Grama, A. Y., Gupta, A. a Kumar, V. 1993. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology.* 1993.

Gropp, Lusk a Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition).* Cambridge : MIT Press, 1999.

Hill, M. D. 1990. What is scalability? *Computer Architecture News.* 1990.

Huss-Lederman, a další. 1998. *MPI: The Complete Reference - Vol 1 The MPI Core.* Cambridge : MIT Press, 1998.

Lewis, T. G. a El-Rewini, H. 1992. *Introduction to Parallel Computing.* Englewood : Prentice-Hall, 1992.

MERKLE, Daniel. 2012. Parallel Computing (Fall 2011). *IMADA.* [Online] 14. 1 2012. Dostupné z WWW: http://www.imada.sdu.dk/~daniel/.

mpi-forum.org. 2013. MPI Documents. *Message Passing Interface Forum.* [Online] 2013. [Citace: 30. 4 2013.] http://www.mpi-forum.org/docs/docs.html.

NERSC. 2013. NERSC: National Energy Research Scientific Computing Centre. [Online] 2013. [Citace: 20. 4 2013.] https://www.nersc.gov/.

Nicol, D. M. a F. H., Willard. 1988. Problem size, parallel architecture, and optimal speedup. 1988, Sv. Journal of Parallel and Distributed Computing.

Petersen, W.P. a Arbenz, P. 2004. *Introduction to Parallel Computing - A Practical Guide with Examples in C.* New York : Oxford University Press Inc., 2004. ISBN 0 19 851576 6.

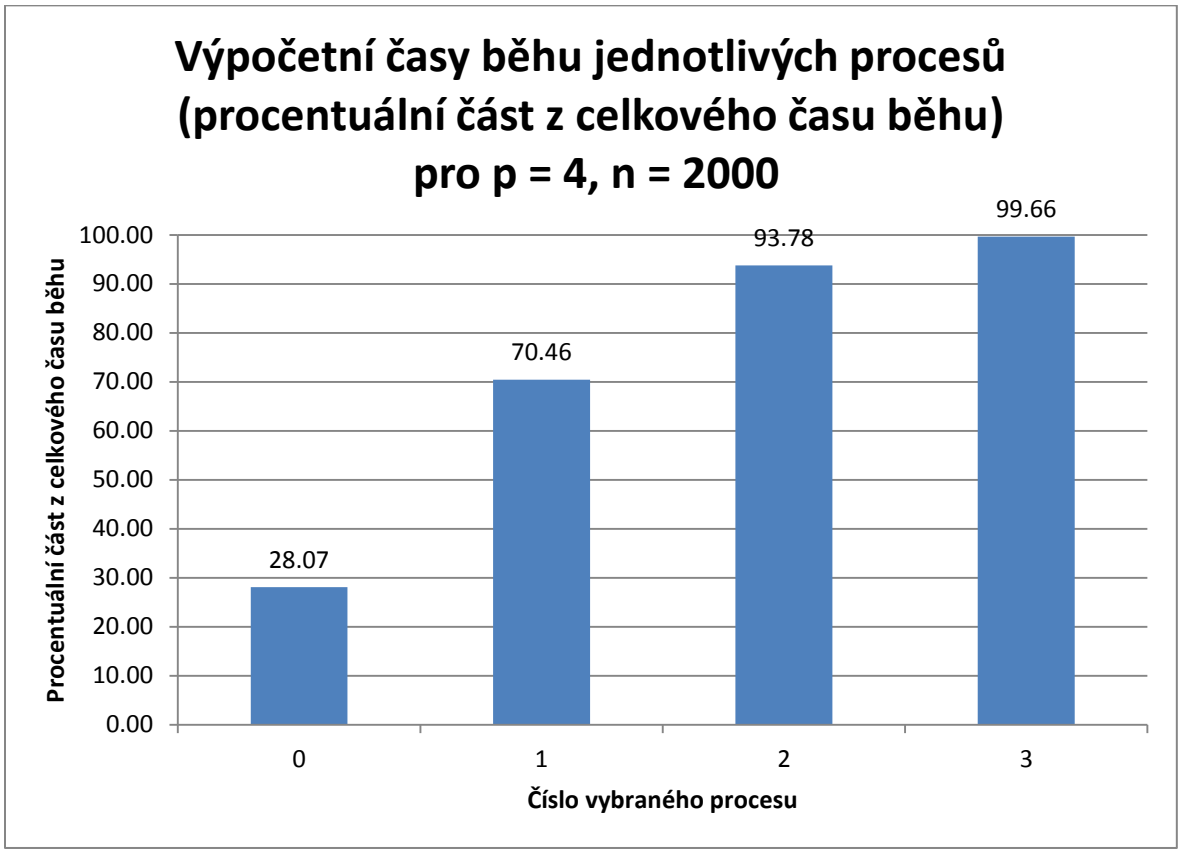
SETI@home. 2013. SETI@home. [Online] University of California, 2013. [Citace: 30. 4 2013.] http://setiathome.berkeley.edu/.

Tanenbaum, Andrew. 2001. *Modern Operating Systems.* 2001. ISBN 0-13-031358-0.

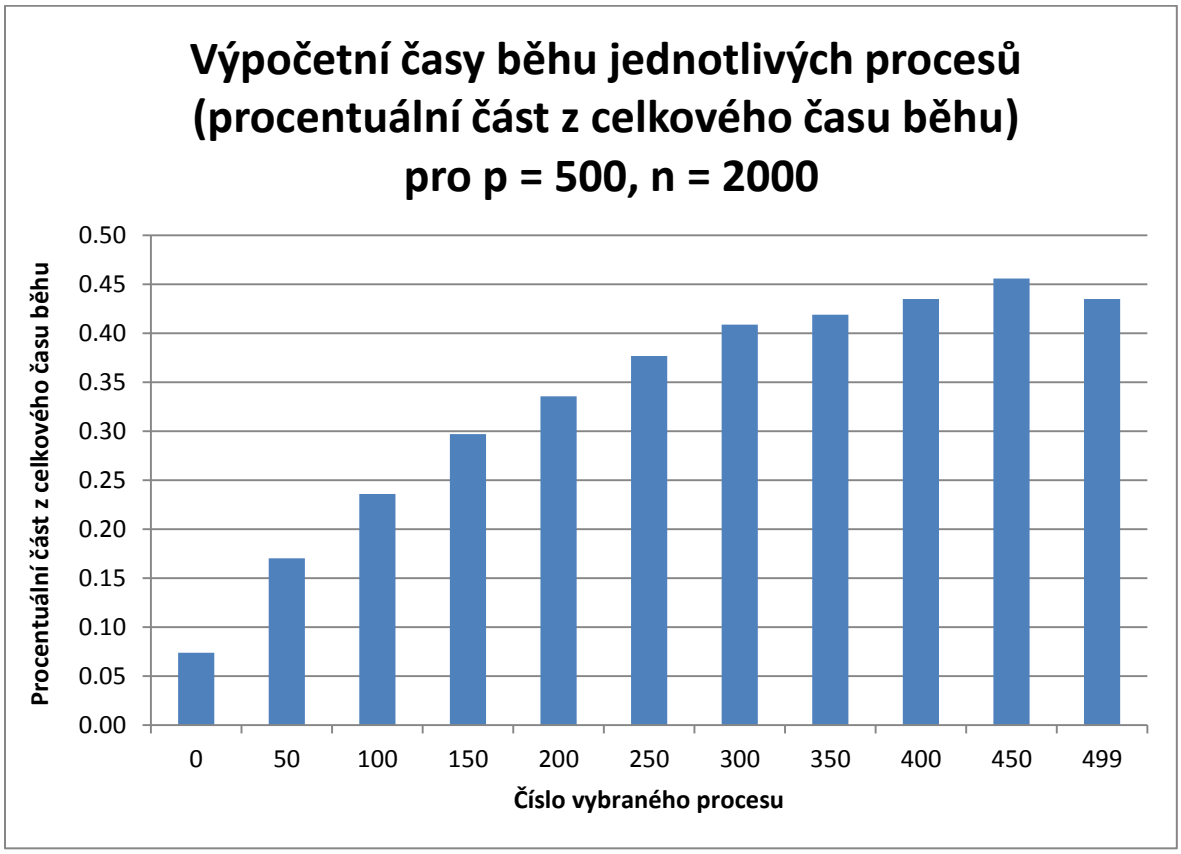
Top500.org. TOP 10 Sites for November 2011. [Online] [Citace: 7. 1 2012.] http://top500.org/lists/2011/11/.

Vitásek, Emil. 1987. *Numerické metody.* Praha : SNTL, 1987.

Příloha A – Časy běhů jednotlivých procesů pro n = 2000 a p = 4



Příloha B – Časy běhů jednotlivých procesů pro n = 2000 a p = 500



Příloha C – Časy běhů jednotlivých procesů pro n = 2000 a p = 1000