VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Testování výkonnosti nástroje Kaira

Performance Tests of the Tool Kaira

2015 Martin Beseda

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Beseda**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Testování výkonnosti nástroje Kaira
Performance Tests of the Tool Kaira

Zásady pro vypracování:

Kaira umožňuje generovat samostatné distribuované MPI/C++ aplikace z vizuálního modelu vycházejícího z Petriho sítí. Hlavním cílem bakalářské práce bude otestovat výkonnost těchto vygenerovaných aplikací v porovnání s podobnými ručně napsanými variantami.

1. Vyberte vhodné aplikace pro testování výkonnosti distribuovaných aplikací. Jako vzor může sloužit například NAS Parallel Benchmarks agentury NASA (http://www.nas.nasa.gov/publications/npb.html).
2. Tyto aplikace realizujte v prostředí C/C++ a MPI. Dle možností jednotlivých aplikací oddělte výkonnou část a část realizující komunikaci v distribuovaném prostředí.
3. Implementujte aplikace znovu, s využitím nástroje Kaira. Při řešení maximálně využijte již existující výkonnou část aplikace. Komunikaci realizujte ve vizuálním modelu nástroje.
4. Proveďte měření, která srovnají různé vlastnosti původních aplikací a aplikací vytvořených v nástroji Kaira.
5. Dá se očekávat, že automaticky vygenerovaná aplikace bude méně výkonná, než člověkem optimalizované řešení. Pokuste se charakterizovat příčiny a navrhněte, možné optimalizace.

Seznam doporučené odborné literatury:

[1] S. Bohm, M. Behalek, O. Meca, and M. Surkovsky. Kaira: Development environment for mpi applications. In Application and Theory of Petri Nets and Concurrency, volume 8489 of Lecture Notes in Computer Science, pages 385-394. Springer International Publishing, 2014.
[2] Kaira: http://verif.cs.vsb.cz/kaira/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7.5.2015 ..... Bureda .....

Zde bych rád poděkoval Ing. Marku Běhálkovi, Ph.D. za návrh tématu mé bakalářské práce, Ing. Václavu Haplovi za konzultace ohledně PETSc a práce s Anselmem a Ing. Stanislavu Bohmovi, Ph.D. za rady ohledně nástroje Kaira. Dále děkuji Matthew Knepleymu, Ph.D. a Hong Zhang, Ph.D. za rady a pohotový vývoj PETSc.

Abstrakt

Prvním cílem této práce jsou otestování rychlosti paralelního programu vygenerovaného nástrojem Kaira. Druhým cílem je potom otestování Kairy pro vývoj v praxi a návrhy vylepšení pro tento nástroj.

Podstatou řešení je implementace Gaussovy eliminační metody pomocí nástroje Kaira a následná implementace řešiče soustav rovnic, který využívá LU-rozklad v PETSc. Výsledek byl získán porovnáním dob běhu testovaných programů.

V této práci bylo experimentálně ověřeno, že program implementovaný v nástroji Kaira je pomalejší, než program psaný čistě sekvenčně, v tomto případě s využitím PETSc.

Přínosem této práce je zjištění efektivity generování kódu nástrojem Kaira a návrhy na zlepšení Kairy pro její použití při vývoji velkých projektů. Tento výstup může být použit pro zlepšení algoritmů generujících paralelní část programu.

Klíčová slova: bakalářská práce, MPI, C, C++, PETSc, Kaira, LU-rozklad, testovací algoritmus

Abstract

The first goal of this thesis is the testing of a parallel program generated by the tool Kaira. The second goal is the testing of Kaira for a large project development and suggestions for this tool.

The essential part of the solution is the implementation of Gaussian elimination in Kaira and the implementation of the equation system solver which uses LU-decomposition, in PETSc.

There was experimentally verified in this thesis, that a program implemented in the tool Kaira is slower, than a similar program implemented sequentially, in this case, using PETSc.

The main benefit of this thesis is the recognition of the effectivity of generating source code by the tool Kaira and suggestions for improvement of the Kaira tool for its application in large projects developement.

Keywords: bachelor thesis, MPI, C, C++, PETSc, Kaira, LU-decomposition, LU-factorization, testing algorithm

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
BLAS	–	Basic Linear Algebra Subprograms
GEM	–	Gaussova eliminační metoda
HPL	–	High-Performance Linpack Benchmark
IPC	–	Inter-Process Communication
LU	–	Lower-Upper
MPI	–	Message Passing Interface
NPB	–	NAS Parallel Benchmarks
PETSc	–	Portable, Extensible Toolkit for Scientific Computation
VSIPL	–	Vector Signal Image Processing Library

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním tématem této bakalářské práce je otestování paralelní implementace LU-rozkladu Gramovy matice v nástroji Kaira a porovnání výkonnosti takto implementovaného programu s paralelní implementací LU-rozkladu v PETSc. Aby se usnadnil vývoj paralelních aplikací, Kaira využívá pro implementaci paralelní části grafický nástroj, který umožňuje vytvářet schémata vycházející z barevných Petriho sítí. Předpokladem tedy je, že za cenu rychlejší a přehlednější implementace algoritmu bude výsledný program pomalejší než kdybychom jej celý implementovali manuálně.

Dalším cílem této práce je ověřit nynější použitelnost nástroje Kaira při vývoji většího projektu a navrhnout vylepšení pro snadnější práci s tímto nástrojem. Předpokladem je, že funkčních chyb bude v této fázi vývoje již minimum, zato však bude nepříliš odladěné rozhraní z pohledu uživatele Kairy. Největšími nedostatky pravděpodobně budou neintuitivní práce s objekty v grafickém nástroji a opomenutí některých funkcionalit „usnadňujícího“ charakteru jako jsou klávesové zkratky nebo problém s orientací v rozsáhlém schématu.

Tato práce je rozdělena do čtyř částí. První část se zabývá možnostmi měření výkonnosti clusterů. Popsány jsou Nas Parallel Benchmarks, HPL benchmark, algoritmus použitý při implementaci v nástroji Kaira i specifika při využití PETSc pro testování v této práci.

Druhá část popisuje použité technologie a algoritmy. Popsáno je PETSc a nástroj Kaira, které se přímo při implementaci využívaly, dále MPI, které je využíváno oběma těmito technologiemi. Vysvětlen je i princip Petriho sítí, ze kterých vychází grafický nástroj Kairy. Z algoritmů je pak popsána Gaussova eliminační metoda, která je základem algoritmu použitého při implementaci v nástroji Kaira, LU-rozklad a Metoda nejmenších čtverců pomocí níž byly generovány matice pro testování.

Ve třetí části je zevrubně popsána implementace obou programů účastnících se testu. Popis programu, který využívá PETSc je doplněn o výpisy zdrojového kódu. Popis programu vytvořeného v nástroji Kaira je pak mimo zdrojových kódů doplněn i o odkazy na přílohu, kde je umístěno schéma Petriho sítě navržené v nástroji Kaira. Schéma umístěno do přílohy z důvodu své velikosti, kterou by narušovalo samotný popis.

Čtvrtá část popisuje výsledky testů na několika velikostech testovacích matic a různých počtech použitých výpočetních uzlů. Výsledky jsou shrnuty v tabulce a pro srovnání obou technologií i vizualizovány grafem.

V závěru jsou pak tyto shrnuty a odůvodněny. Dále jsou popsány možnosti dalšího využití nástroje Kaira i poznatky, které jsem díky této bakalářské práci získal.

2 Měření výkonu clusterů

Clustery, tedy skupiny vzájemně propojených výpočetních uzlů (nodů), se vyznačují velmi vysokým výkonem. Pro měření tohoto výkonu se obvykle využívají benchmarky, ve kterých jsou implementované výpočetně náročné algoritmy. Příkladem těchto algoritmů může být násobení matic, LU-rozklad, řazení extrémně velkých číselných polí atd.

Pro tato testování už existují hotové, profesionální benchmarky, např. NPB - paralelní benchmarky vyvinuté NASA nebo HPL - paralelní implementace Linpack benchmarku pro systémy s distribuovanou pamětí.

2.1 NPB

Sada programů určena pro hodnocení výkonu clusterů, která byla vyvinuta v NASA. Na rozdíl od HPL obsahuje více benchmarků, nespecializuje se jen na jeden typ testovacího algoritmu. Tyto benchmarky jsou odvozeny z aplikací pro simulace proudění kapalin. Jedná se o open-source, zdrojové kódy jsou psány v C a Fortranu. Benchmarky jsou rozděleny do několika skupin, podle cílového využití a jejich jednotlivé verze do tříd, podle výpočetní náročnosti implementovaných testů [36].

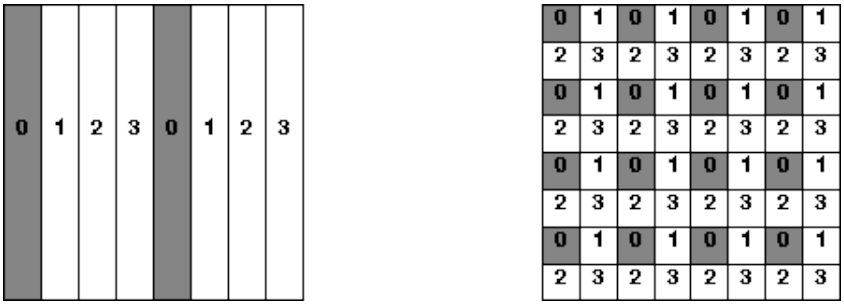
V současnosti je aktuální verze 3.x. Verze 1.x byla především rozborem algoritmů pro testování clusterů, benchmarky nebyly určeny pro testování v praxi. Stěžejní byla teoretická specifikace. Později, s rozvojem počítačů, přestaly stačit třídy benchmarků A a B a byla implementována třída C pro měření v praxi. Tento doplněk vyšel spolu se specifikací verze 2. Verze 2.2 pak obsahovala implementaci nových benchmarků, verze 2.3 byla první kompletní implementací v MPI, byla v ní zavedena i nová třída W. Ve verzi 2.4 byla zavedena nová třída benchmarků D. Ve verzi 3 byly přidány implementace benchmarků z verze 2.x v OpenMP, Javě a High Performance Fortran. Ve verzích 3.1 a 3.2 byly přidány nové benchmarky, ve verzi 3.3 byla přidány třídy E a F pro extrémně náročné testy. Největší změnou ve verzi 3.x je přidání nového typu benchmarků - NPB-Multi-Zone, které testují víceúrovňovou a hybridní paralelizaci, zvláště zaměřenou na kombinaci MPI/OpenMP. [30, 31, 32, 33, 34, 35]

Kódy benchmarků jsou volně dostupné na internetových stránkách projektu [37].

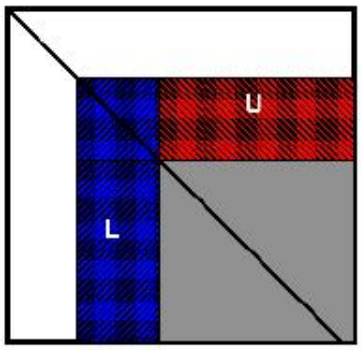
2.2 HPL

Softwarový balík, který řeší náhodný hustý lineární systém na 64 bitech, tedy v tzv. *double precision*. V balíku jsou i zabudované nástroje pro měření času výkonnosti clusteru, pro měření tedy nejsou třeba žádné další nástroje. Samotný balík HPL ale potřebuje k provozu, aby bylo na clusteru nainstalováno MPI pro rozesílání dat mezi nody a jeden z matematických balíků pro lineární algebru BLAS nebo VSIPL. V současnosti je aktuální verze 2.1.

Algoritmus využívá 2-D blok-cyklickou distribuci, čímž dosahuje rovnoměrnějšího vyvážení zátěže procesorů, než u 1-D distribucí „do sloupců“ (viz obrázek 1). Pro LU-rozklad byla zvolena „right-looking“ varianta, ukázka rozdělení matice v jedné iteraci v této variantě je na obrázku 2. [29]



Obrázek 1: Ukázka cyklické distribuce dat mezi procesy 1-D (vlevo) a 2-D (vpravo) (zdroj [28])



Obrázek 2: Ukázka „right-looking“ verze algoritmu pro LU-rozklad (zdroj [28])

Všechny informace o balíku HPL i jeho soubory ke stažení jsou dostupné z internetových stránek projektu [28].

2.3 Metoda měření v této práci

Cílem této práce není testování výkonnosti samotného clusteru, ale programu vytvořeného v nástroji Kaira, v porovnání s programem napsaným v PETSc. Oba tyto testy byly spuštěny na clusteru Anselm. Podrobný popis testu a výsledků je uveden v sekci 7 , str. 40. Pro oba testy byl jako algoritmus zvolen LU-rozklad, vzhledem k tomu, ze se jedná o jeden z nejpoužívanějších způsobů, jak testovat clustery a výsledný program je pak využitelný i jako součást řešiče lineárních rovnic.

Pro měření času byl v nástroji Kaira využit *system_clock* z C++ třídy *chrono* [23]. V PETSc byl využit nativní měřič, který se dá spustit z terminálu pomocí přepínače *-log_summary*.

Podrobně je LU-rozklad popsán v sekci 3.2.

2.3.1 Kaira

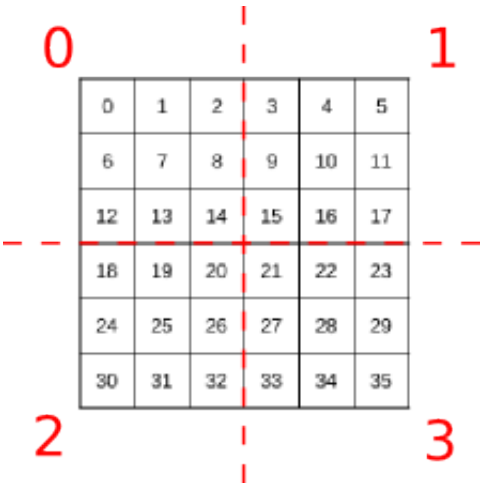
Pro test jsem zvolil variantu Gaussovy eliminační metody (podrobně popsána v sekci 3.1) pro zjištění horní trojúhelníkové matice, tj. stěžejní části LU-rozkladu. GEM je velmi rozšířený a všeobecně známý algoritmus, takže je velmi vhodný pro demonstrativní účely, dále je poměrně snadno implementovatelný a paralelizovatelný.

Detaily implementace jsou popsány v sekci 4.2.

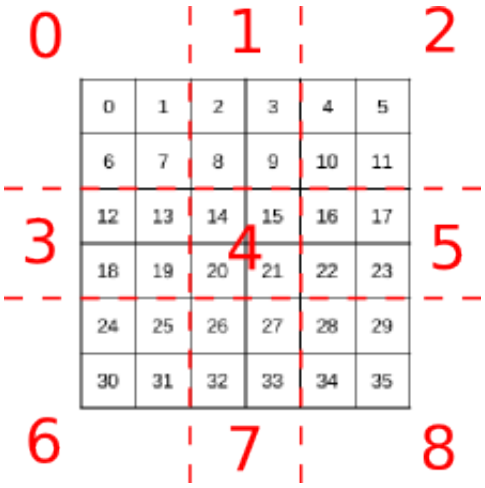
Distribuce dat je v tomto případě realizována formou 2-D statické distribuce, tj. faktorizovaná matice je rozdělena do stejně velkých čtverců a tyto jsou přiděleny procesům v *procesní mřížce*. Proces má pevně přidělenou jen svoji část a po jejím vyřešení a rozeslání dat dalším procesům zůstává nečinný po zbytek doby běhu programu. Tento způsob byl zvolen z důvodu jednoduchosti implementace v porovnání s dynamickou distribucí. Statický způsob ale také funguje na různých počtech procesů - podmínkou je, aby bylo možné rozdělit faktorizovanou matici do rovnoměrných čtverců, které pak připadnou jednotlivým procesům (vyjádřeno vztahem 1). Porovnání distribuce matice 6x6 na 4 a 9-ti procesech je znázorněno na obrázcích 3 a 4.

$$\boxed{\frac{Pocet\_prvku\_na\_hrane\_matice}{\sqrt{Pocet\_procesu}} \in R} \tag{1}$$

2.3.1.1 Procesy v mřížce lze rozdělit do 4 tzv. *procesních rolí* (viz obrázek 5 , str. 9). Podle své role pak procesy čekají na data od jiných procesů, řeší pomocí nich svoji část matice a pak přeposílají dál vlastní data. Po každé iteraci řešení se procesní role posouvají (viz obrázek 6 , str. 9), tedy každý proces v roli *Common* se jednou stane *Rootem*, *Row 0* nebo *Col 0*.



Obrázek 3: Distribuce mezi 4 procesy



Obrázek 4: Distribuce mezi 9 procesů

Poznámka 2.1 Na obrázcích 5 a 6 je červenou barvou značena role *Root*, modrou role *Row 0*, zelenou role *Col 0* a bílá nevybarvená pole jsou role *Common*. Šedá pole jsou pak

již pasivní, resp. vyřešenou částí matice, kde jsou procesy pro zbytek běhu programu neaktivní.

Obrázek 5: Role na počátku programu

Obrázek 6: Role za běhu programu

2.3.1.2 Rozesílaná data jsou informace potřebné k průběhu GEM na více procesech. Vzhledem k tomu, že procesy v distribuovaném systému „nevidí“ celou matici, jako ve sdílené paměti, ale jen její část, potřebují potřebné informace zaslat. Tyto informace jsou trojího typu:

LineLeaders jsou čísla na začátku řádků, tedy hodnoty ve sloupci pod aktuálním *Multiplicatorem*.

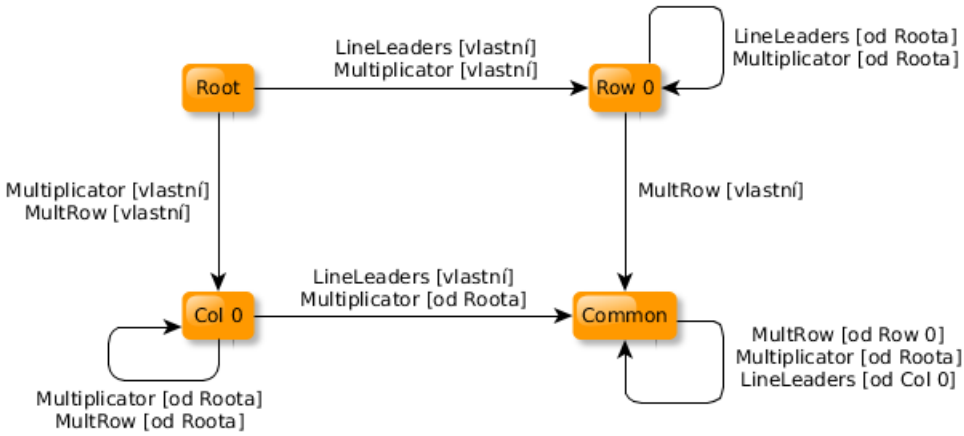
Příkladem můžou být hodnoty 1, 7 a 3 v prvním kroku úprav matice v příkladu 3.4 , str. 13.

Multiplicator je číslo v root procesu, kterým se násobí zbytek matice mimo *MultRow*, pozice pod ním se v téže iteraci řešení nulují. Je to hodnota na pozici „nejvíce vlevo a nahoře“ v aktivní (zatím nevyřešené) části matice.

Příkladem *root prvku* může být číslo 1 v prvním kroku úprav matice v příkladu 3.4 , str. 13.

MultRow je řádek, který začíná hodnotou *Multiplicator*. Je to tedy řádek nejvýše umístěný v aktivní části matice a jeho příkladem můžou být čísla 1, 3 a 5 z prvního kroku příkladu 3.4 , str. 13.

Každý jednotlivý proces tedy musí načíst svou část z matice, dle potřeby ji vynásobit multiplikátorem, a pak odečíst násobky řádků od MultRow tak, aby na pozicích pod hlavní diagonálou zůstaly nuly a výsledky každého z těchto kroků ve výpočtu také musí poslat dalším procesům. Možnost zpoždění některých výpočtů je řešena pomocí kontrolních tokenů, kdy se nemůže výpočet v procesu aktivovat, pokud mu od obou procesů „příjemců“ nepřijdou kontrolní tokeny, že už jsou na stejné iteraci jako on. Před začátkem



Obrázek 7: Diagram rozesílání dat mezi rolemi procesů

další iterace si každý proces v roli *Common* ověřuje, jestli už je na něm řada se změnou role. Princip tohoto algoritmu z pohledu jednotlivého procesu je popsán schématem na obrázku 8.

Poznámka 2.2 Účelem diagramu na obrázku 8 je ilustrovat základní princip fungování algoritmu, ne jeho detailní popis. Nejsou v něm tedy uvedena konkrétní data, která se odesílají z procesu (to je ilustrováno schématem na obrázku 7, str. 10) ani není detailně popsáno čekání a odesílání kontrolních tokenů (to je podrobně popsáno v sekci 4.2 , str. 31).

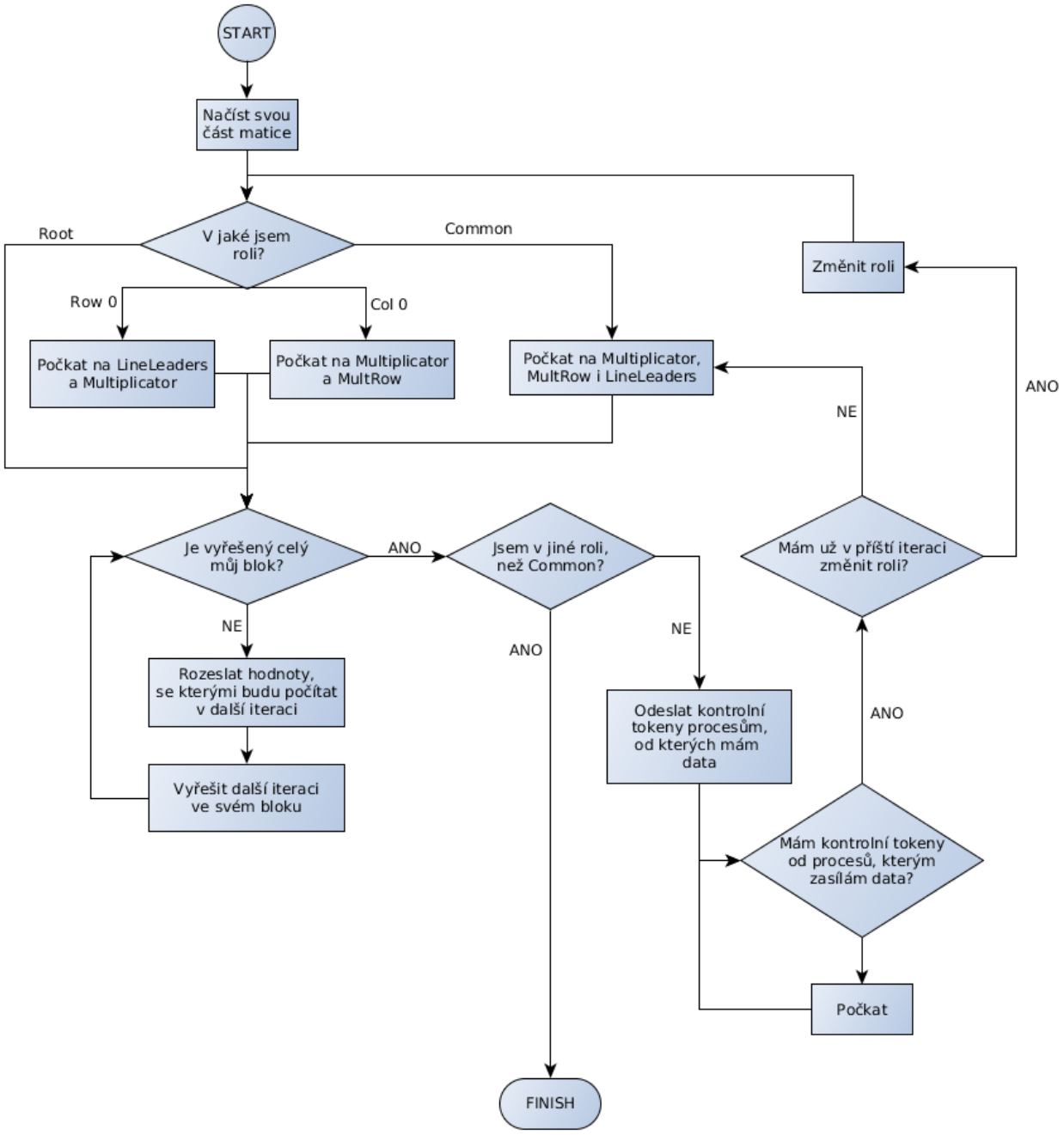
2.3.2 PETSc

Pro srovnání s implementací v nástroji Kaira byl zvolen program napsaný v jazyku C s využitím knihovny PETSc. PETSc bylo zvoleno díky tomu, že je vyvíjeno v Argonne National Laboratory, stejně jako *MPICH* a je tedy syntakticky přizpůsobeno pro práci s MPI, které také využívá. Dalším důvodem bylo to, že PETSc disponuje vysokoúrovňovým rozhraním k řešení matematických problémů, takže implementace LU-rozkladu husté matice je rychlá a poměrně jednoduchá.

Pro měření času byl použit nativní nástroj PETSc, který se dá vyvolat z terminálu parametrem *-log_summary*.

Detaily použitého algoritmu v PETSc nejsou podstatou této práce a nejsou tedy popsány, implementace programu v PETSc je uvedena v sekci 4.1 , str. 28. Stručné shrnutí matematického aparátu využitého pomocí PETSc je uvedeno v sekci 3.7.1 , str. 25.

Informace o MPI jsou v sekci 3.4 , str. 18. MPICH konkrétně je popsáno v sekci 3.4.2. PETSc je popsáno v sekci 3.7 , str. 25.



Obrázek 8: Vývojový diagram paralelizované GEM z pohledu jednotlivého procesu

3 Použité technologie a algoritmy

V této sekci jsou popsány technologie i matematické postupy použité při implementaci. Popis jednotlivých „položek“ je krátký, zaměřený na nejvýznačnější znaky. Cílem je čtenáři představit nebo připomenou popisovanou technologii, resp. algoritmus, nikoliv jej rozebrat do detailu. Pro získání detailnějších informací lze využít uvedené zdroje.

3.1 Gaussova eliminační metoda

Gaussova eliminační metoda je metoda pro řešení soustav lineárních rovnic. Výsledkem této metody je soustava v tzv. „schodovém“ tvaru, tedy horní trojúhelníková matice. Tato je velmi výhodným výsledkem, protože se dá snadno využít v dalších matematických postupech, např. pří výpočtu determinantu matice, inverzní matice nebo při opakovaném řešení soustavy rovnic s různými vektory pravé strany. Horní trojúhelníková matice je také základem pro LU-rozklad (popsán v 3.2 , str. 14), při kterém se běžně Gaussova eliminační metoda využívá.

3.1.1 Elementární řádkové úpravy

Gaussova eliminační metoda umožňuje využívat množinu ekvivalentních úprav, které nezmění výslednou hodnotu neznámých v soustavě lineárních rovnic. V některých případech je pak třeba provést ještě „kompenzační aktivity“. Klasickým případem takovéto situace jsou úpravy *prohození řádků* a *vynásobení řádku konstantou* při výpočtu determinantu. V prvním případě se změní znaménko determinantu a v případě lichého počtu prohození řádků pak musíme ve výsledku znaménko změnit. Ve druhém případě, při násobení řádku matice konstantou je touto konstantou vynásobený i determinant a v posledním kroku musíme tento vydělit stejnou konstantou (viz příklad 3.7 , str. 15).

Vynásobení řádku matice konstantou Každý řádek matice můžeme vynásobit libovolnou konstantou. Násobíme tak všechny koeficienty v dané lineární rovnici a hodnota výsledku se tedy nezmění.

Příklad 3.1
Vynásobení 2. řádku matice číslem 5

$$\begin{pmatrix} 1 & 3 & 5 \\ 7 & 2 & 0 \\ 3 & 2 & 4 \end{pmatrix} \quad | \cdot 5 \quad \sim \begin{pmatrix} 1 & 3 & 5 \\ 35 & 10 & 0 \\ 3 & 2 & 4 \end{pmatrix}$$

■

Přičtení násobku řádku k jinému řádku matice Tato úprava je stejným postupem, který se používá u běžné sčítací metody - odečítají se hodnoty v jednom řádku vynásobené konstantou od hodnot v jiném řádku. Tímto postupem se dá matice upravit tak,

aby v dolní trojúhelníkové části zůstávaly jen nuly, tedy aby vznikla *horní trojúhelníková matice*.

Příklad 3.2
Přičtení 3-násobku 1. řádku ke 3. řádku

$$\begin{pmatrix}1 & 3 & 5\\7 & 2 & 0\\3 & 2 & 4\end{pmatrix} \begin{matrix}\rceil^{3}\\ \\ \hookleftarrow_{+}\end{matrix} \sim \begin{pmatrix}1 & 3 & 5\\7 & 2 & 0\\6 & 11 & 19\end{pmatrix}$$

■

Prohození řádků matice Poslední ekvivalentní úpravou je prohození řádků matice. Tato úprava je užitečná např. v případě, kdy některý řádek již obsahuje 0 a stačí jej tedy prohodit s jiným, abychom se přiblížili trojúhelníkové matici.

Příklad 3.3
Prohození 2. a 3. řádku

$$\begin{pmatrix}2 & 5 & 8\\0 & 0 & 4\\0 & 8 & 6\end{pmatrix} \begin{matrix}\\ \hookleftarrow\\ \hookleftarrow\end{matrix} \sim \begin{pmatrix}2 & 5 & 8\\0 & 8 & 6\\0 & 0 & 4\end{pmatrix}$$

■

Příklad 3.4
Zjištění horní trojúhelníkové matice pomocí Gaussovy eliminační metody.

$$\begin{pmatrix}1 & 3 & 5\\7 & 2 & 0\\3 & 2 & 4\end{pmatrix} \begin{matrix}\rceil^{-7}\ \rceil^{-3}\\ \hookleftarrow_{+}\\ \hookleftarrow_{+}\end{matrix} \sim \begin{pmatrix}1 & 3 & 5\\0 & -19 & -35\\0 & -7 & -11\end{pmatrix} \begin{matrix}\\ \\ \mid \cdot -19\end{matrix} \sim \begin{pmatrix}1 & 3 & 5\\0 & -19 & -35\\0 & 133 & 209\end{pmatrix} \begin{matrix}\\ \rceil^{-7}\\ \hookleftarrow_{+}\end{matrix} \sim$$

$$\sim \begin{pmatrix}1 & 3 & 5\\0 & -19 & -35\\0 & 0 & -36\end{pmatrix}$$

■

3.1.2 Využití v této práci

Gaussova eliminační metoda v jednoduché formě byla použita při implementaci LU-rozkladu, resp. zjištění horní trojúhelníkové matice, v nástroji Kaira. Tento algoritmus byl zvolen kvůli své jednoduchosti a velké popularitě. Části této práce zabývající se samotným výpočtem v nástroji Kaira, který využívá Gaussovu eliminaci, tedy bez větších obtíží pochopí i člověk, který se Petriho sítěmi, Kairou ani lineární algebrou nezabývá více do hloubky.

3.2 LU-rozklad

Matematická metoda pro získání horní (**U**pper) a dolní (**L**ower) trojúhelníkové matice ze čtvercové matice A, kdy platí vztah daný rovnicí 2.

$$\boxed{A = LU} \tag{2}$$

Tato metoda byla vyvinuta Alanem M. Turingem, který ji poprvé uvedl v publikaci [21]. Její využití je významné např. při opakovaném řešení rovnic s jiným vektorem pravé strany. Dále je důležitým krokem při určování determinantu matice. Podrobnější popis těchto případů je popsán níže v této sekci.

Pokud je na pozici matice a_{00} číslo 0, znamená to, že alespoň jeden z prvků l_{00} a u_{00} se rovná 0. To naznačuje, že trojúhelníková matice s nulou může mít determinant 0, tj. být singulární. Když je matice A regulární, L ani U nesmí být singulární, tedy se tato situace řeší prohozením řádků, aby na pozici [0,0] bylo nenulové číslo. Tomuto postupu se říká *částečná pivotace* a bývá realizována vynásobením tzv. *permutační matice*, která se značí P, maticí A. Tento postup je ilustrován příkladem 3.5.

Příklad 3.5

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 0 & 2 & 7 \\ 5 & 3 & 1 \\ 4 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 5 & 3 & 1 \\ 0 & 2 & 7 \\ 4 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.8 & -0.2 & 0.2 \end{pmatrix} * \begin{pmatrix} 5 & 3 & 1 \\ 0 & 2 & 7 \\ 0 & 0 & 18 \end{pmatrix}$$

■

Částečná pivotace obecně se tedy dá zapsat vztahem, který je uveden v rovnici 3.

$$\boxed{PA = LU} \tag{3}$$

V některých případech je nutno dále permutovat sloupce výsledné matice po součinu P*A, což realizujeme vynásobením této matice další *permutační maticí*, značenou Q. Tento postup je označován jako *kompletní pivotace*. Tento postup je ilustrován příkladem 3.6.

Příklad 3.6

$$\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix} * \begin{pmatrix}0&2&7\\5&3&1\\4&2&3\end{pmatrix} * \begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix} = \begin{pmatrix}1&3&5\\7&2&0\\3&2&4\end{pmatrix} =$$
$$= \begin{pmatrix}1&0&0\\7&1&0\\3&\frac{7}{19}&\frac{1}{19}\end{pmatrix} * \begin{pmatrix}1&3&5\\0&-19&-35\\0&0&36\end{pmatrix}$$

Poznámka 3.1 Tento příklad je jen ilustrační, při praktickém výpočtu by v tomto případě byla dostačující částečná pivotace.

■

Kompletní pivotace se dá tedy zapsat vztahem, který je uveden v rovnici 4.

$$\boxed{PAQ = LU} \tag{4}$$

3.2.1 Časté případy použití LU-rozkladu

3.2.1.1 Určení determinantu Pro určení determinantu je třeba dostat matici do trojúhelníkového tvaru a pak vynásobit hodnoty na *hlavní diagonále*. Hlavní diagonálou se rozumí prvky matice v rozmezí $A_{00}..A_{nn}$. Hodnota součinu se však liší, pokud se prohodí řádky během úprav matice nebo nějaký řádek vynásobí konstantou.

Příklad 3.7
Výpočet determinantu z trojúhelníkové matice.

$$[1 * (-19) * (-36)] \div (-19) = 684 \div (-19) = -36$$

Poznámka 3.2 Při výpočtu trojúhelníkové matice byl ve druhém kroku 3. řádek vynásoben číslem -19 (viz příklad 3.4), je jím tedy třeba vydělit i součin čísel na hlavní diagonále.

■

3.2.1.2 Řešení rovnic je jedním z nejdůležitějších případů využití LU-rozkladu. Ten umožňuje, aby se rovnice dala řešit velmi rychle pro tytéž levé strany rovnic, resp. jejich matici, s různými vektory prav= strany $\overrightarrow{b}$.

Základní vztah je uveden v rovnici 5, kde A je soustava rovnic zapsaná maticí a vektor *x* je „vektorem neznámých“.

$$\boxed{A\overrightarrow{x} = \overrightarrow{b}} \tag{5}$$

Řešení samotné se skládá ze dvou hlavních kroků - dopředné a zpětné substituce. [22]

Dopředná substituce využívá vztahu zapsaného rovnicí 6.

$$\boxed{\overrightarrow{y} = L^{-1}\overrightarrow{b}} \tag{6}$$

Zpětná substituce využívá vztahu zapsaného rovnicí 7.

$$\boxed{U\overrightarrow{x} = \overrightarrow{y}} \tag{7}$$

Oba tyto vztahy se dají odvodit, resp. dokázat následujícím postupem:

$$\begin{aligned} A\overrightarrow{x} &= \overrightarrow{b} \\ LU\overrightarrow{x} &= \overrightarrow{b} \\ U\overrightarrow{x} &= L^{-1}\overrightarrow{b} \end{aligned}$$

$$\begin{aligned} U\overrightarrow{x} &= \overrightarrow{y} \\ \overrightarrow{y} &= L^{-1}\overrightarrow{b} \end{aligned}$$

Příklad 3.8
Řešení soustav rovnic, kde je levá strana totožná a mění se jen vektor pravé strany $\overrightarrow{b}$. Levá strana je zapsána maticí $\begin{pmatrix} 3 & 2 \\ 24 & 21 \end{pmatrix}$ a vektory pravé strany jsou (7, 66) a (17, 156).

Matice i s vektorem pravé strany jsou tedy zapsány takto:

$$\left(\begin{array}{cc|c} 3 & 2 & 7 \\ 24 & 21 & 66 \end{array}\right)$$

$$\left(\begin{array}{cc|c} 3 & 2 & 17 \\ 24 & 21 & 156 \end{array}\right)$$

LU-rozklad matice popisující levou stranu rovnic:

$$L : \begin{pmatrix} 1 & 0 \\ 8 & 1 \end{pmatrix}$$

$$U : \begin{pmatrix} 3 & 2 \\ 0 & 5 \end{pmatrix}$$

Invertovaná spodní trojúhelníková matice.

$$L^{-1} = \begin{pmatrix} 1 & 0 \\ -8 & 1 \end{pmatrix}$$

Až do tohoto kroku byl výpočet tentýž pro oba vektory pravé strany, nemusel se tedy provádět pro každý vektor pravé strany zvlášť.

Následující části výpočtu jsou již pro každý vektor pravé strany individuální. Výpočet patřící k vektoru $\overrightarrow{b}(7, 66)$ je označen jako *a)*, výpočet patřící k vektoru $\overrightarrow{b}(17, 156)$ je označen jako *b)*.

Dopřednou substitucí se získá vektor $\overrightarrow{y}$:

$$a)\begin{pmatrix} 1 & 0 \\ -8 & 1 \end{pmatrix} * \begin{pmatrix} 7 \\ 66 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix} \qquad b)\begin{pmatrix} 1 & 0 \\ -8 & 1 \end{pmatrix} * \begin{pmatrix} 17 \\ 156 \end{pmatrix} = \begin{pmatrix} 17 \\ 20 \end{pmatrix}$$

„Vektor neznámých“ značený $\overrightarrow{x}$ se získá zpětnou substitucí:

$$a)\begin{pmatrix} 3 & 2 \\ 0 & 5 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix} \qquad b)\begin{pmatrix} 3 & 2 \\ 0 & 5 \end{pmatrix} * \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 17 \\ 20 \end{pmatrix}$$

Hodnoty neznámých pro vektor $\overrightarrow{b}(7, 66)$ jsou tedy 1 a 2, pro vektor $\overrightarrow{b}(17, 156)$ pak 3 a 4. ■

3.2.2 Využití v této práci

V této práci je LU-rozklad využitý jako algoritmus, který je implementován různými metodami pro srovnání výkonu takto vzniklých aplikací. Vzhledem k zaměření této práce na testy nástroje Kaira byly jako testovací data zvoleny Gramovy matice generované Metodou nejmenších čtverců (sekce 3.3, str. 17). Tyto matice nikdy nemají na pozici [0,0] nulu a tak nebylo třeba řešit pivotaci, která by podstatně znesnadnila implementaci v nástroji Kaira.

LU-rozklad byl pro tento účel zvolen proto, že je jedním z typických algoritmů, na kterých se tyto testy běžně provádějí a je pro jeho implementaci se dají využít známé postupy, jako např. Gaussova eliminační metoda (sekce 3.1, str. 12). Další výhodou volby LU-rozkladu je jeho časté použití v praxi, takže i čtenář, který se nespecializuje na lineární algebru, snadno pochopí části této práce, které se zabývají popisem tohoto algoritmu.

3.3 Metoda nejmenších čtverců

Jde o aproximační, matematicko-statistickou metodu, která byla vyvinuta Carlem Fridrichem Gaussem. Metoda je určena k řešení tzv. *přeurčených soustav rovnic*. Takto se nazývají soustavy, kde je více rovnic, než neznámých. Podstatou metody je minimalizace součtů čtverců odchylek vůči rovnici.

Tato metoda má mnoho využití, hlavním je minimalizace chyb z naměřených hodnot, např. u měření signálů, geodetických měření atd. [43]

3.3.1 Využití v této práci

V této práci byla Metoda nejmenších čtverců použita pro generování testovacích matic. Byla použita proto, že výsledné matice jsou symetrické, takže je možné je rozložit jak Choleského, tak LU-rozkladem.

Algoritmus 1 Metoda nejmenších čtverců (zdroj [43])

Input: $f, (\varphi_0, ..., \varphi_n)$
  for $i := 0, ..., n$ **do**
    for $j := 0, ..., n$ **do**
      $[A]_{i,j} := (\varphi_i, \varphi_j)$
    end for
    $[b]_i := (f, \varphi_i)$
  end for$[c_0, ..., c_n]^T :=$ *c je řešení soustavy* $Ac = b$
Output: $c_0, ..., c_n$

3.4 MPI

MPI je specifikace knihoven poskytujících API pro přeposílání zpráv mezi procesy v systémech se sdílenou pamětí i bez ní [1]. MPI knihovny využívají stejnojmenný protokol, který je z pohledu modelu ISO/OSI zasazen do relační vrstvy. Obvyklým transportním protokolem je TCP.

Hello, world napsaný v MPI je ve výpisu 1 (str. 20).

3.4.1 Verze API pro programovací jazyky

Implementace MPI API existují pro více programovacích jazyků, nejčastěji používané jsou v C, C++ a Fortranu. Dále je MPI implementováno např. pro Javu [2] nebo Python [3].

3.4.2 Implementace MPI standardu

Nejvíce používanými implementacemi MPI knihoven jsou MPICH a OpenMPI.

MPICH je první implementací standardu MPI 1.x, byl vyvinut v Argonne National Laboratory (viz [4]) ve spolupráci s Mississippi State University. Argonne pokračuje ve vývoji i nadále, současná verze MPICH implementuje standard MPI 3.x [5].

OpenMPI je open-source implementace standardu MPI. Tato implementace vychází z několika starších, převážně pak z LAM/MPI, FT-MPI a LA-MPI s cílem využít silných stránek každé z těchto technologií [6] [7].

3.4.3 Programátorský přístup v MPI API

MPI API nenabízí nic jako „sdílenou schránku“, přístup vychází čistě z posílání jednotlivých zpráv mezi procesy. Hlavními prostředky, které tento přístup využívá, jsou komunikátory, funkce pro point-to-point komunikaci a funkce pro kolektivní komunikaci.

Pro upřesnění libovolného typu zpráv (z pohledu programátora, tedy např. „dopravní prostředek“, „časy příjezdu“, nikoliv „integer“ nebo „string“) je u těchto funkcí celočíselný (integer) parametr, tzv. tag.

Uživatelem využívaný paměťový prostor kde jsou proměnné programu, se nazývá *aplikační buffer* [9].

Komunikátory představují množiny procesů v běžící MPI aplikaci. Lze je vytvářet dynamicky za běhu. Vždy je k dispozici speciální komunikátor MPI_COMM_WORLD, který obsahuje všechny procesy běžící v aplikaci. Ten je vytvářen automaticky při spuštění programu a existuje po celou dobu jeho běhu [8].

Point-to-point komunikace je realizována funkcemi, které čekají na data od konkrétního procesu (resp. je na něj zasílají) s konkrétním tagem. Tyto funkce se dělí na *blokující* a *neblokující* [9].

Blokující funkce se provedou, až je to bezpečné vzhledem k zápisu do *aplikačního bufferu*. Zástupci této skupiny jsou např. funkce *MPI_Send* a *MPI_Recv*.[10]

Neblokující funkce se provedou téměř ihned, nečekají na korektní dokončení žádných událostí - fungují tak, že „pověří“ MPI knihovnu, aby potřebné činnosti vykonala ihned, jakmile to bude možné. Zástupci této skupiny jsou např. funkce *MPI_Isend* a *MPI_Irecv*. [11]

Kolektivní komunikace je realizována funkcemi, které se vždy chovají jako *blokující*. Zástupci této skupiny jsou např. funkce *MPI_Bcast*, *MPI_Gather* nebo *MPI_Scatter*. Tyto funkce vždy přeposílají data všem procesům obsaženým v zadaném *komunikátoru*. [12]

3.4.4 Využití v této práci

V této práci je MPI využito nepřímo prostřednictvím knihovny PETSc, kde se využívá jeho implementace MPICH pro jazyk C a v nástroji Kaira, kde se generuje MPI kód podle Petriho sítě navržené v grafickém nástroji. Kaira využívá MPICH nebo OpenMPI, v této práci se i u Kairy používá MPICH.

```
#include <mpi.h>
#include <stdio.h> // Kvuli definovani konstanty NULL

int main() {
    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    printf ("Hello, world! My PID is: %d !\n", world_rank);

    MPI_Finalize();
}
```

Výpis 1: Hello, world v jazyce C s využitím MPI

3.5 Petriho sítě

Petriho sítě (PetriNets, PN) představují matematický nástroj pro modelování a simulaci diskrétních systémů (např. systémů hromadné obsluhy apod.) [27].

Byly navrženy Carlem Adamem Petrim. Využívají 3 základní objekty - *místa*, *přechody* a *hrany*. Zvláštním prvkem jsou pak *tokeny*, v češtině také někdy označované jako *tečky* nebo *značky*. Ty představují přeposílaná data a značí se tečkou, resp. vyplněným kroužkem.

Příklad Petriho sítě je na obrázku 9.

3.5.1 Části

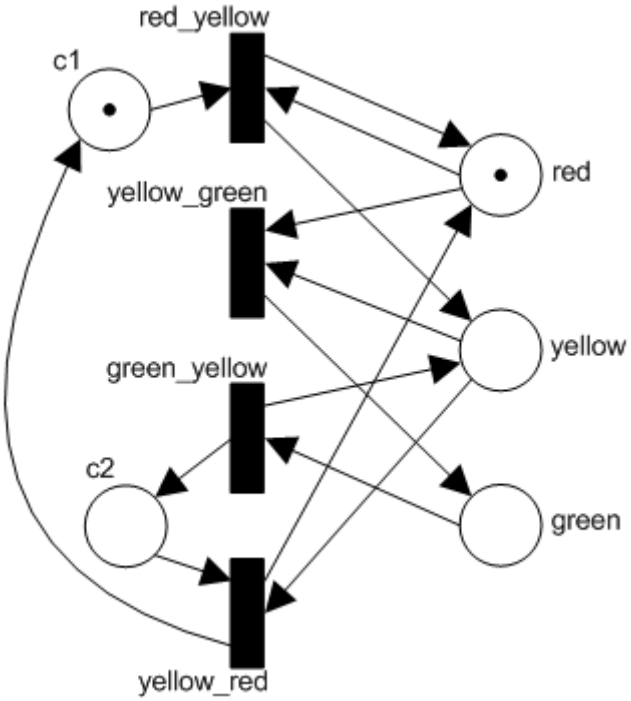
Místo je prvkem Petriho sítě, ve kterém se nacházejí *tokeny*. Místo samo slouží jen k ukládání tokenů, jinak je statickým prvkem v síti. Značí se kroužkem.

Přechod je prvkem Petriho sítě, který slouží k aktivaci hrany, resp. k přenosu tokenů po hraně do dalšího stavu. Značí se obdélníkem. Má 2 stavy - *uschopněný*, resp. *aktivovaný* a *neuschopněný*. Přechod se uschopní tehdy, když je na každém místě na jeho vstupních hranách alespoň jeden token.

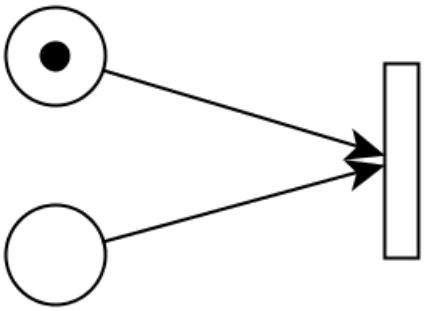
Toto chování je znázorněno na obrázcích 10 a 11 , str. 21.

Když je proces uschopněný, může se *odpálit*. Odpalování je nedeterministické, tedy se můžou odpálit všechny uschopněné přechody v daném kroku nebo se nemusí odpálit žádný.

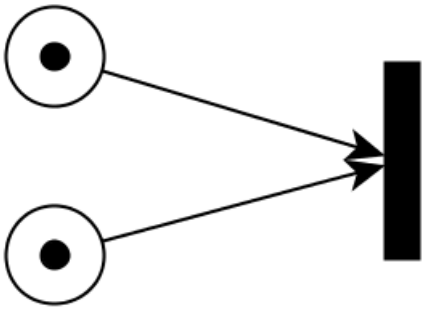
Hrana je objekt v Petriho síti, který znázorňuje spojení mezi *místem* a *přechodem*. Hrany jsou orientované, tokeny se tedy nemohou přenášet libovolným směrem. Hrana může spojovat jen místo a přechod, nikdy dvě místa nebo dva přechody. Ve schématu je hrana značena šipkou.



Obrázek 9: Příklad Petriho sítě (zdroj [38])



Obrázek 10: Ukázka neuschopněného přechodu



Obrázek 11: Ukázka uschopněného přechodu

3.5.2 Druhy

Pro lepší popis specifických systémů bylo vyvinuto mnoho druhů Petriho sítí. Pro ilustraci je zde popsáno několik příkladů.

C/E sítě C/E znamená „Condition / Event“, tedy podmínka a událost. C/E sítě jsou původním modelem Petriho sítí, místa se zde chápou jako podmínky (pokud je v místě token, je splněná) a přechody jako události (když se splní všechny podmínky pro událost, ta může proběhnout).

P/T sítě P/T znamená „Place / Transition“, tedy místo a přechod. Z tohoto modelu pochází současná terminologie. Místa jsou zde chápána jako stavy systému a přechody představují změny těchto stavů. V P/T modelu byla oproti C/E přidána možnost explicitního určení maximální kapacity místa a také možnost určení počtu tokenů pro přesun po určité hraně. V případě, že se explicitně tyto hodnoty neurčí, je kapacita místa chápána jako nekonečná a tokeny se budou přesouvat po hraně po jednom.

Časované sítě Mohou mít „časovanou“ kteroukoliv ze svých částí. Existují tedy sítě s časovanými místy, přechody, hranami i tokeny.

Časování v případě míst funguje tak, že token čeká určitou dobu na místě, než může být použit.

V případě přechodů se doba průběhu přechodu přizpůsobí časování.

Časování u hrany určuje, jakou dobu po ní bude token přenášen.

Časování u tokenů je opatří „razítkem“, které říká, kdy může být token znovu použit.

Barevné sítě Vychází z P/T sítí, je možné mít více *typů tokenů* (tzv. „barvy“), místům je přiřazena *třída*, tj. typ tokenů, který se zde bude nacházet. Jedno místo může mít takto i více tříd. Přechodům může být dále přiřazena podmínka, tzv. *guard*, po jehož splnění může být přechod teprve „uschopněn“. Každé hraně je dále přiřazen *hranový výraz*, který se skládá z proměnných a konstant. Ten po vyhodnocení reprezentuje množinu, resp. multimnožinu (prvky se mohou opakovat) tokenů stejné třídy, jakou má jejich původní místo.

Z tohoto typu Petriho sítí vychází i varianta sítě použitá pro nástroj Kaira.

3.5.3 Využití v této práci

Petriho sítě jsou v této práci využity nepřímo prostřednictvím nástroje Kaira. Ten ve svém grafickém nástroji umožňuje vytvářet diagramy vycházející z barevných Petriho sítí a princip práce s tímto nástrojem je tak velmi podobný návrhu čisté Petriho sítě. Díky svému grafickému debug režimu Kaira umožňuje sledovat v tomto grafickém nástroji průběh programu doslova „krok po kroku“, takže může uživatel nejen sledovat běh v Petriho síti, ale dokonce i sám ovlivňovat, které přechody se odpálí, se kterým procesem konkrétně se má nyní pracovat atd. Schéma vycházející z Petriho sítí tak využívá jejich

snadné čitelnosti a jednoduchého značení a obohacuje jej o možnosti použití funkcí na hranách, přenášení konkrétních dat místo obecných tokenů atd.

Konkrétní popis práce s prvky Petriho sítí v nástroji Kaira jsou popsány v sekci 3.6.1, str. 23.

3.6 Kaira

Kaira je open-source vývojové prostředí, které má za cíl spojit prototypování, implementaci i debugging aplikace pro paralelní systémy do jednoho nástroje [13]. Pro tyto účely byla vyvinuta koncepce, kdy část programu řešící IPC je implementována v grafickém nástroji, kde vývojář navrhne schéma vycházející z *barevných Petriho sítí*. [14, 15] Kódy projektu Kaira jsou volně přístupné v GitHub repozitáři [16].

3.6.1 Programátorský přístup v nástroji Kaira

Hlavní devizou Kairy z s ohledem na inovativní přístup je grafický nástroj. V tomto může vývojář navrhnout Petriho síť přizpůsobenou pro účely tohoto nástroje. Výsledkem je pak schéma, které je zároveň funkční implementací IPC a dá se použít k přehlednému ladění systému, kdy vývojář vidí, které tokeny jsou přenášeny po hranách, který přechod je v daném kroku aktivní, jaká data se nacházejí na daném místě atd. V situaci, kdy je simulace spuštěna pro více procesů se dají prohlížet i data patřící konkrétnímu procesu, stejně jako provádět kroky jen pro vybrané procesy. To je velmi užitečné v případě, že je potřeba otestovat jen určitou situaci a „krokovat“ simulaci tak dlouho, dokud neproběhnou potřebné kroky na testovaných procesech, zatímco by probíhaly i na jiných, by bylo velmi zdlouhavé.

Pro samotnou funkcionalitu na místech a přechodech pak jde psát sekvenční kód v jednoduchém vestavěném editoru. Editovatelný je vždy jen blok funkce, zbytek kódu je automaticky generovaný. Konkrétní specifika kódu u míst a přechodů jsou popsány níže. Ukázka Petriho sítě i sekvenčního kódu v nástroji Kaira je přiložen k příkladu 3.9.

Pro vkládání hlavičkových souborů, psaní funkcí pro serializaci procesů (str. 23 v odstavci *Tokeny*) nebo jiných globálně dostupných funkcí se využívá editovatelný hlavičkový soubor.

Tokeny v nástroji Kaira představují konkrétní přenášená data. Mají svůj explicitně zadaný datový typ a v grafickém nástroji Kaira disponují náhledem na přenášená data. U datových typů, které nelze automaticky serializovat (vlastní struktury, třídy atd.) je nutné tento náhled určit pomocí funkce *token_name* [19]. Pro samotnou serializaci je pak nutné napsat funkce *pack* a *unpack* [20].

Hrany po kterých se tokeny přeposílají mohou mít v popisku i jednoduché operace s nimi (např. inkrementace) a volání nativních funkcí, popř. funkcí vytvořených v kódu hlavičkového souboru projektu. Dále jde zadat přeposlání dat jinému procesu pomocí operátoru @.

Místa mohou obsahovat sekvenční kód, popř. explicitně zadanou hodnotu pro počáteční inicializaci. Kód v nich obsažený se provede ihned po spuštění programu, na rozdíl od *přechodů* nečekají na žádný vstup.

V parametrech k vygenerované funkci v sekvenčním kódu je vždy parametr *ctx* typu Context, což je třída, která umožňuje zjistit ID procesu, počet procesů atd. a je vždy přístupná jak v sekvenčních kódech, tak i v grafickém nástroji. Dalším parametrem je parametr *place* typu TokenList<T>, pomocí jehož metody *add* můžeme přidávat nové hodnoty do TokenListu daného místa, tedy je vytvářet z pohledu grafického nástroje.

Příklad kódu místa je ve výpisu 3.

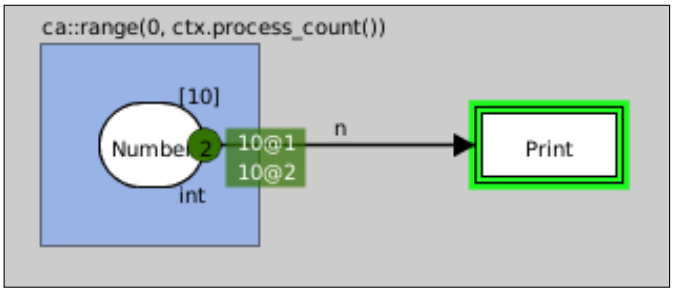
Přechody mohou také obsahovat sekvenční kód, který se ale, na rozdíl od míst, provede až tehdy, když se jsou k dispozici tokeny na všech vstupních hranách, resp. místech. Přechody lze spouštět opakovaně, vždy, když mají k dispozici vstupy.

V parametrech k vygenerované funkci je vždy parametr *ctx* stejně jako u *místa*, dalším parametrem pak může být struktura *var* typu Vars. Ta obsahuje proměnné reprezentující všechny vstupy do přechodu.

Příklad kódu přechodu je ve výpisu 2.

Příklad 3.9
Pokud potřebujeme inicializovat místo nějakou hodnotou pro všechny procesy, můžeme použít *inicializační oblast* - v grafickém nástroji Kairy jako modrý obdélník. Tomu pak lze explicitně říct, pro které procesy inicializace platí.

Na obrázku 12 můžeme vidět síť, která inicializuje tokeny v místě „Number“ pro všechny procesy celočíselnou hodnotou (integer) 10. Přechod „Print“ pak pro jednotlivé procesy načítá tokeny s integerem a vypisuje Hello, world hlášku spolu se svým PID a přijatou hodnotou, v tomto případě „10“. Kód tohoto přechodu je ve výpisu 2.



Obrázek 12: Petriho síť v nástroji Kaira

```
struct Vars {
    int &n;
};

void transition_fn (ca::Context &ctx, Vars &var)
{
    std::cout << "Hello, world! My PID is "
              << ctx.process_id()
               << " and I've just received number "
              << var.n
              << "!"
              << std::endl;
}
```

Výpis 2: Sekvenční kód přechodu "Print"

■

Využití v této práci

Nástroj Kaira je hlavním tématem této práce, byl použit pro implementaci Gaussovy eliminační metody pro získání horní trojúhelníkové matice, ze kterého byl naměřen čas pro srovnání s implementací v PETSc. Níže jsou v této práci uvedeny návrhy pro zlepšení Kairy, zvláště z pohledu uživatelského rozhraní a intuitivnosti práce.

```
void place_fn(ca::Context &ctx, ca::TokenList<int> &place)
{
    place.add(6);
}
```

Výpis 3: Příklad kódu místa

3.7 PETSc

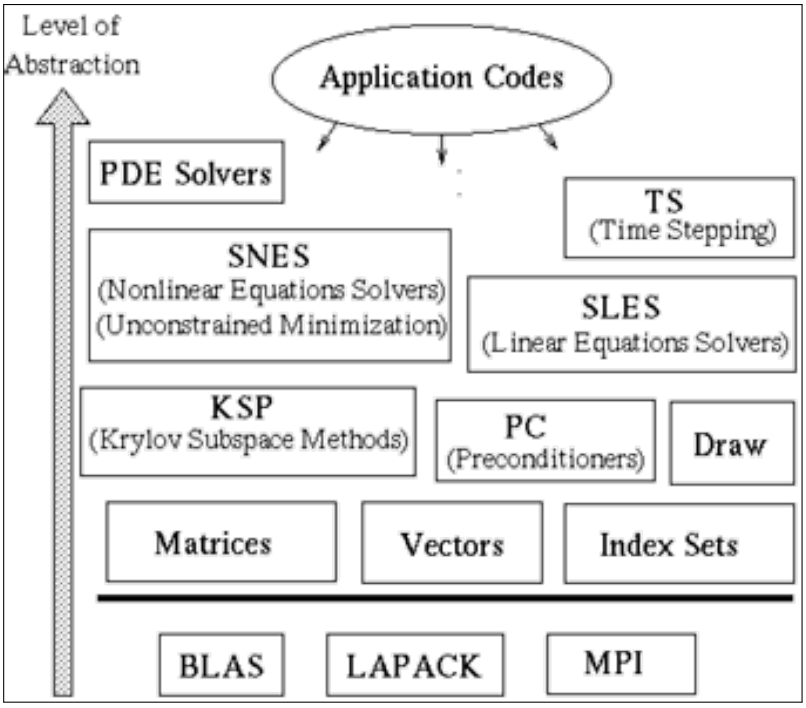
PETSc je sada nástrojů a knihoven pro vědecké výpočty [17].

PETSc využívá pro meziprocesní komunikaci MPI, konkrétně implementaci MPICH. MPICH i PETSc jsou vyvíjeny v Argonne National Laboratory. Využívají tedy podobných principů, např. funkce *MPI_Initialize* a *Petsc_Initialize* nebo *MPI_Finalize* a *Petsc_Finalize* jsou z pohledu programátora využívajícího tyto knihovny de facto totožné.

PETSc je ale, na rozdíl od různých implementací MPI vysokoúrovňovým nástrojem určeným pro rychlé řešení složitých matematických problémů. Jeho základní struktura je ukázána na obrázku 13.

3.7.1 Popis solverů a předpodmiňovačů

Hlavními částmi PETSc jsou *řešiče* (neboli *solvery*) a předpodmiňovače. PETSc disponuje jak non-lineárními, tak i lineárními iteračními řešiči, které využívají předpodmiňovače. V PETSc dále existuje i možnost využít *direct solveru*. Vlastnosti se dají těmto součástem



Obrázek 13: Schéma základní struktury PETSc (zdroj [18])

nastavovat buď z terminálu pomocí parametrů (např. *-ksp_type cg*) nebo pomocí funkcí ve zdrojovém kódu (např. *KSPSetType()*). V této práci je použita druhá možnost.

KSP je třída umožňující řešení lineárních soustav. [39] V této práci byl použit řešič *KSPPREONLY*, který obsahuje pouze akci předpodmiňovače.

PC je rozhraní pro použití *předpodmiňovačů*. PETSc poskytuje přímé řešiče *PCCHOLESKY* (kompletní Choleského rozklad) a *PCLU* (kompletní LU-rozklad). Také obsahuje jejich nekompletní varianty *PCICC* a *PCILU*, které lze použít jako předpodmiňovače iteračních metod. Dalšími příklady mohou být *PCEISENSTAT* nebo *PCJACOBI*. [40, 42]

V této práci byl použit předpodmiňovač *PCLU*. PETSc jej poskytuje jen v sekvenčním provedení, PC ale disponuje možností využít externí balíky. Pro paralelní zpracování kompletního LU-rozkladu byl tedy využit v kombinaci s *PCLU* i externí balík Elemental. [41]

3.7.2 Využití PETSc v této práci

PETSc v této práci bylo využito pro implementaci řešiče soustavy lineárních rovnic zapsaných jednou maticí. Tento řešič díky snadnému použití KSP, tedy Krylovových metod

v PETSc, dokáže řešit nejen LU-rozklad, ale přímo i soustavy rovnic, což umožňuje větší variabilitu v testech nástroje Kaira.

Dále bylo PETSc využito pro implementaci generátoru matic v binárním formátu. Tento binární formát je originální formát PETSc a je konvenčním formátem pro práci s maticemi uloženými v externích souborech, proto bylo nutné využít PETSc i v kódu, který pro samotné generování matice PETSc nepotřebuje.

4 Implementace testů

V této sekci jsou popsány detaily obou implementací. Pro PETSc je popis doplněn zdrojovým kódem, u Kairy jsou přidány i reference na jednotlivé části schématu sítě. Ta pro svou velikost je uvedena v příloze A.

4.1 PETSc

V této sekci je popsána implementace v PETSc. Popsány budou význačné části kódu, které přímo souvisí s řešením LU-rozkladu a soustavy matic - použité knihovny, výpisy, ověřování vstupů atd. tedy uvedeny nejsou.

4.1.1 Vstupy

Cesta k souboru s maticí je prvním parametrem. Jedná se o textový řetězec, resp. v C o pole datového typu *char*.

Počet řešení soustavy rovnic je druhým parametrem. Jedná se o datový typ integer. Tento parametr byl přidán pro rozšíření možností testování. Řešená soustava se skládá z načtené matice a automaticky vygenerovaného vektoru stejného rozměru, jako matice, inicializovaného číslem 1.0 (viz výpis 7).

4.1.2 Popis zdrojového kódu

Prvním krokem je inicializace PETSc pomocí funkce *PetscInitialize()*. Makro *CHKERRQ()* vrací v případě chyby hodnotu typu *PetscErrorCode*, která je zde uložena do proměnné *ierr*.

```
ierr = PetscInitialize ( &argc, &args, (char*)0, help ); CHKERRQ( ierr );
```

Výpis 4: Inicializace PETSc

Soubor s maticí je otevřen pomocí „binárního prohlížeče“, tj. pomocí funkce *PetscViewerBinaryOpen()*. Pro matici je deklarována proměnná *A* typu *Mat*. Typ matice je na počátku určen funkcí *MatSetFromOptions()*. Proměnná matice je inicializována daty ze souboru funkcí *MatLoad()*. Vzhledem k tomu, že PCLU neumí paralelně zpracovávat výpočty, byl pro tento účel zvolen externí balík Elemental. Pro použití s tímto balíkem je nutná matice ve formátu MATELEMENTAL. Toto přetypování je provedeno funkcí *MatConvert()*. Potom už není *prohlížeč*, resp. viewer potřeba, takže je dealokován funkcí *PetscViewerDestroy()*.

```
ierr = MatCreate( PETSC_COMM_WORLD, &A ); CHKERRQ( ierr );
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
ierr = PetscViewerBinaryOpen( PETSC_COMM_WORLD, args[1], FILE_MODE_READ, &
    viewer ); CHKERRQ( ierr );
ierr = MatLoad( A, viewer ); CHKERRQ( ierr );
ierr = MatConvert( A, MATELEMENTAL, MAT_REUSE_MATRIX, &A ); CHKERRQ( ierr );
```

```
ierr  = PetscViewerDestroy( &viewer ); CHKERRQ( ierr );
```

Výpis 5: Vytvoření a inicializace matice ze souboru

Pro KSP solver je deklarována proměnná *ksp* typu *KSP*. Solver samotný je vytvořen funkcí *KSPCreate()*. Matici, se kterou má pracovat a druhou, kterou má použít při konstrukci předpodmiňovače určuje funkce *KSPSetOperators()*. Obě „pozice", tedy definici lineárního systému i konstrukci předpodmiňovače zastává tatáž matice určená vstupem programu. Funkcí *KSPSetType()* se nastaví typ řešiče. V této práci je zadán *KSPPREONLY*, což je „degenerovaný" řešič, který využívá jen jednu aplikaci předpodmiňovače (PC). Tento typ KSP se používá v kombinaci s přímými řešiči, jako je *PCLU* (viz níže).

Pro předpodmiňovač je deklarována proměnná *pc* typu *PC*. Samotný předpodmiňovač je vytvořen funkcí *KSPGetPC()*. Jeho typ je pak nastaven funkcí *PCSetType()* na *PCLU*. *PCLU* je přímý řešič, který využívá kompletní LU-rozklad (viz sekce 3.7.1 , str. 25).

Protože *PCLU* je v samotném PETSc jen v sekvenční implementaci, pro paralelní funkcionalitu bylo nutné zvolit balík, pro který PETSc poskytuje wrapper a lze tedy použít přes jednotné rozhraní. Pro tento účel byl vybrán balík Elemental, který je v kódu použit předáním parametru *MATSOLVERELEMENTAL* funkci *PCFactorSetMatSolverPackage()*.

Nastavený KSP solver je „aktivován" pomocí funkce *KSPSetUp()*. V tomto případě se provede samotný LU-rozklad.

```
ierr  = KSPCreate( PETSC_COMM_WORLD, &ksp ); CHKERRQ( ierr );
ierr  = KSPSetOperators( ksp, A, A ); CHKERRQ( ierr );
ierr  = KSPSetType( ksp, KSPPREONLY ); CHKERRQ( ierr );
ierr  = KSPGetPC( ksp, &pc ); CHKERRQ( ierr );

ierr  = PCSetType( pc, PCLU ); CHKERRQ( ierr );

ierr  = PCFactorSetMatSolverPackage( pc, MATSOLVERELEMENTAL ); CHKERRQ( ierr );

ierr  = KSPSetUp( ksp ); CHKERRQ( ierr );
```

Výpis 6: Vytvoření a nastavení KSP a PCLU

Pomocí funkce *MatCreateVecs()* jsou alokovány vektory kompatibilní s maticí A. Vektory jsou uloženy do proměnných x a y typu *Vec*. Vektor x slouží jako „vektor neznámých" $\overrightarrow{x}$, vektor y jako „vektor pravé strany" $\overrightarrow{b}$ (viz příklad 3.8 , str. 17). Kompatibilitou je pak myšlena ekvivalentní distribuce mezi procesy tak, aby bylo možné provést násobení $Ax = y$. Vektor y je inicializován hodnotou 1.0 pomocí funkce *VecSet()*.

Řešení se provede v počtu opakování, jaký byl zadán na vstupu. Samotné řešení soustavy je zprostředkováno funkcí *KSPSolve()*.

```
ierr  = MatCreateVecs( A, &x, &y ); CHKERRQ( ierr );
ierr  = VecSet(y, 1.0) ;  CHKERRQ( ierr );

for( i = 0; i < atoi(args[2]) ; i++ ) {
    ierr  = KSPSolve( ksp, y, x ) ;  CHKERRQ( ierr );
}
```

Výpis 7: Vytvoření pravé strany soustavy a řešení

KSP solver, matice i vektory jsou dealokovány pomocí funkcí *KSPDestroy()*, *MatDestroy()* a *VecDestroy*.

```
ierr = KSPDestroy( &ksp ); CHKERRQ( ierr );
ierr = MatDestroy( &A ); CHKERRQ( ierr );
ierr = VecDestroy( &x ); CHKERRQ( ierr );
ierr = VecDestroy( &y ); CHKERRQ( ierr );
```

Výpis 8: Dealokace prostředků

PETSc je ukončeno funkcí PetscFinalize().

```
ierr = PetscFinalize(); CHKERRQ( ierr );
```

Výpis 9: Ukončení PETSc

4.1.3 Výstup

Výstupem programu je log vyvolaný přepínačem *-log_summary* [42]. Pro práci podstatnou informací jsou pak položky *MatLUFactorSym* a *MatLUFactorNum*. Ukázka části logu je na obrázku 14.

```
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------

--- Event Stage 0: Main Stage

ThreadCommRunKer       1 1.0 4.7684e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
ThreadCommBarrie       1 1.0 3.0994e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatSolve               1 1.0 2.6239e-02 1.0 4.50e+01 1.0 0.0e+00 0.0e+00 0.0e+00 13 35  0  0  0  13 35  0  0  0     0
MatLUFactorSym         1 1.0 5.9605e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatLUFactorNum         1 1.0 4.6983e-02 1.0 8.33e+01 1.0 0.0e+00 0.0e+00 0.0e+00 22 65  0  0  0  22 65  0  0  0     0
MatAssemblyBegin       1 1.0 3.3379e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatAssemblyEnd         1 1.0 3.0994e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatLoad                1 1.0 7.6294e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
KSPSetUp               1 1.0 6.9141e-06 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
KSPSolve               1 1.0 2.6389e-02 1.0 4.50e+01 1.0 0.0e+00 0.0e+00 0.0e+00 13 35  0  0  0  13 35  0  0  0     0
PCSetUp                1 1.0 4.7803e-02 1.0 8.33e+01 1.0 0.0e+00 0.0e+00 0.0e+00 23 65  0  0  0  23 65  0  0  0     0
PCApply                1 1.0 2.6261e-02 1.0 4.50e+01 1.0 0.0e+00 0.0e+00 0.0e+00 13 35  0  0  0  13 35  0  0  0     0
VecSet                 4 1.0 4.1008e-05 1.0 0.00e+00 0.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
------------------------------------------------------------------------------------------------------------------------
```

Obrázek 14: Ukázka části PETSc logu

4.1.4 Možné optimalizace

Hlavní slabinou implementace v PETSc je využití formátu MATELEMENTAL pro zpracovávanou matici. Funkce pro tento formát jsou stále ve vývoji a nejsou tedy optimalizované, a jsou velmi pomalé při paralelním zpracování velkých matic. V této práci je tedy takovým „úzkým hrdlem“ funkce *MatConvert()* (viz sekce 4.1.2, str. 28).

Dalším potenciálním „úzkým hrdlem“ je funkce *MatLoad()*, která při načítání matice ve formátu MATELEMENTAL načte matici ve formátu MATDENSE a následně volá opět funkci *MatConvert()*. Jediným současným řešením této situace by tedy bylo generování matice za běhu jako je popsáno v sekci 4.2.4 na straně 35.

4.2 Kaira

V této části je popsána implementace v nástroji Kaira. Vzhledem k rozměrům schématu popisujícího IPC budou popsány jen jeho význačné části.

4.2.1 Vstupy

Počet procesů je prvním parametrem. Jeho datový typ je integer.

Velikost strany matice je druhým parametrem. Datový typ je integer. Mezi velikostí strany a počtem procesů musí platit vztah 1 (str. 8).

Název souboru s maticí je třetím parametrem. Datový typ je string. Matice v souboru musí být uložena v ASCII formátu, kdy po každém čísle následuje mezera a řádky v souboru odpovídají řádkům v matici.

4.2.2 Popis schématu a zdrojového kódu

Na začátku se spustí funkce *get_block_size()*, která zjistí, jak velký blok hodnot bude patřit každému procesu. Pokud není splněna podmínka daná rovnicí 1 (str. 8) a nelze tedy vytvořit stejně velký blok pro každý proces, funkce *get_block_size()* vrátí -1 a program se s upozorněním ukončí.

Potom se uloží do proměnných hodnoty jako ID procesu, počet procesů, velikost matice, počet bloků v jednom řádku procesní mřížky (viz obrázek 3 na str. 8) a souřadnice bloku procesu z pohledu celé matice.

K práci se souborem je využita třída *fstream* [44]. Pomocí její metody *is_open()* je ošetřen případný neúspěch při otevírání souboru s maticí. V takovém případě se program ukončí s chybovou hláškou.

Samotné načtení dat do bloku procesu probíhá tak, že každý proces prochází soubor a kontroluje, jestli je již na „svém prostoru“. To zjistí výpočtem z indexu levého horního prvku svého bloku (proměnné *block_begin_X* a *block_begin_Y*) a velikosti bloku (proměnná *block_size*).

Načtená matice se předá do konstruktoru třídy *mat* [46] z knihovny Armadillo [45]. Matice se tak nekopíruje v paměti do nového umístění (parametr *copy_aux_mem*), ale „naváže“ se na ni wrapper a je tak k dispozici rozhraní pro operace s touto maticí, jako např. násobení řádků konstantou 3.1.1 nebo transpozice.

Jako *token* do *místa* v grafickém nástroji se přidá pomocí metody *add* objektu *place*, který je třídy *TokenList*. 3.6.1

```
int block_size = get_block_size(ctx);
    if ( block_size == -1 ) {
        cout << "ERROR: Spatne zadany pocet procesu k velikosti matice!" << endl;
        ctx.quit();
    } else {
        int process_id = ctx.process_id();
        int process_count = ctx.process_count();
```

```
    int matrix_size = param::size();
    int blocks_per_line = int(sqrt(process_count));
    int block_begin_X = (process_id % blocks_per_line) * block_size;
    int block_begin_Y = floor(process_id / blocks_per_line) * block_size;

    double *matrix_part = new double[block_size * block_size];
    fstream file (param::matrixFile());
    string line;
    int matrix_part_counter = 0;

    if( file.is_open()) {
        for(int row = 0; getline( file, line); row++) {
            vector<string> nums = split(line);
            int nums_len = nums.size();

            for(int col = 0; col < nums_len; col++) {
                if( row >= block_begin_Y
                        && row < (block_begin_Y + block_size)
                        && col >= block_begin_X
                        && col < (block_begin_X + block_size) ) {

                    matrix_part[matrix_part_counter++] = stringToDouble(nums[col]);
                }
            }
        }
    } else {
        cout << "ERROR opening mat file" << endl;
        ctx.quit();
    }

    file.close();

    arma::mat arma_matrix_part( matrix_part, block_size, block_size, false, true );
    place.add(arma_matrix_part);
}
```

Výpis 10: Načtení matice ze souboru

Rozesílání dat (viz sekce 2.3.1.2 , str. 9) je realizováno pomocí *transitních míst* (viz obrázek 22 , str. 49, označeno jako „New Transite_***“), kam se posílají tokeny jiným procesům a odkud je tyto „adresované“ procesy vyzvedávají. Jednotlivé přechody se tedy neuschopní (viz sekce 3.5.1, str. 20), dokud nebudou mít na transitních místech všechny tokeny potřebné k provedení iterace řešení.

Přeposílání samotné se realizuje v každé iteraci výpočtu na jednotlivém procesu v *Compute přechodu* (viz obrázek 19 , str. 46). V případě, že jsou hodnoty „převzaté“, tedy např. proces v roli *Common* (viz sekce 2.3.1.1 , str. 8) přijal *lineLeaders* (viz sekce 2.3.1.2 , str. 9) od jiného procesu v roli *Col 0*, tak se hodnoty taktéž pouze předají dál, tj. pošlou se na *transitní místo*.

V situaci, kdy je proces „autorem“ těchto hodnot, tedy je v roli *Root*, *Col 0* nebo *Row 0*, funguje přeposílání tak, že zašle token s hodnotou na místo patřící k příslušnému „vytvářecímu přechodu“ (např. místo *lineLeaders* a přechod *Create col Transite*, obr. 19,

22, str. 46, 49) a na druhé místo, k němu patřící, zašle číslo iterace řešení v daném bloku, označené jako *k*. Z těchto hodnot je pak vytvořen *transitní objekt*, který tak nese informaci nejen o datech, ale i o tom, z kolikáté iterace řešení pocházejí. Tokeny na „vytvářecím" přechodu má již cílový proces, nikoliv odesílatel. Příklad transitní třídy je ve výpisu 11.

```
class Transite_col {
private:
    arma::colvec m;
    int k;

public:
    Transite_col();
    Transite_col( arma::colvec p_m, int p_k );

    arma::colvec getM() const;
    int getK() const;

    void setM( arma::colvec p_m );
    void setK( int p_k );
};
```

Výpis 11: Třída Transite_col

Samotný výpočet bloku matice je realizován sekvenčním kódem zapsaným v přechodu *Compute*. Vychází z klasické Gaussovy eliminace (viz sekce 3.1, str. 12) a to tak, že jakmile zná *Multiplicator*, vynásobí všechny příslušné řádky svého bloku. Příslušné řádky jsou všechny, mimo toho, na němž leží *Multiplicator*. Příklad tohoto vynásobení je uveden v prvním cyklu *for*, ve výpisu 12.

Dalším krokem ve výpočtu je odečtení hodnoty v řádku s *Multiplicatorem*, vynásobené hodnotou *lineLeader* od právě počítaného řádku. Tímto postupem se dosáhne „vynulování" hodnot pod *Multiplicatorem*, takže se matice dostává do „schodovitého tvaru". Když je tento úplný, tedy když jsou pod *hlavní diagonálou* jen nuly, je výpočet dokončen a výsledkem je *horní trojúhelníková matice*. Tento postup je předveden v druhém cyklu *for* ve výpisu 12. Iterace řešení je označena *k*. Tato hodnota zároveň odpovídá číslu řádku s aktivním *Multiplicatorem*.

```
if (ctx.process_id() == 0) {
    var.begin_time = std::chrono::system_clock::now();
}

int size = param::size();
int block_size = get_block_size(ctx);

double mult = var.om( var.k, var.k );

arma::colvec lineLeaders = var.om.col(var.k);

for( int i = var.k+1; i < block_size; i++ ) {
    var.m.row(i) *= mult;
}

for( int i = var.k+1; i < block_size; i++ ) { // radky
```

```
            for( int j = var.k; j < block_size; j++ ) { // sloupce
                var.m.row(i)(j) -= var.m.row(var.k)(j) * lineLeaders[i];
            }
        }
    };
```

Výpis 12: Výpočet bloku matice v ROOT roli

Procesní role (viz 2.3.1.1, str. 8) jsou představovány kódy v *Compute* přechodech (viz obr. 19, str. 46). Změna procesní role je tedy implementována jako přesun tokenu obsahujícího zpracovávaný blok matice na jiný *Compute* přechod v rámci jednoho procesu. Rozhodování o této změně se provádí po každém dokončeném *cyklu řešení*, tj. vynulování všech hodnot pod jedním *Multiplicatorem*. Toto rozhodnutí je provedeno pomocí přechodu *Decide role change* a funkce *change_role()* (viz obr. 20, str. 47). Kód této funkce je ve výpisu 13.

```
bool change_role(ca::Context &ctx, int solution_num) {
    // V pripade, ze je proces na diagonale nebo v dolni
    // casti matice, staci pockat pocet odpaleni prechodu,
    // ktery se rovna poctu sloupcu pred aktualnim procesem
    // - v techto pripadech jej kazdy root "zasahne"
    //
    // Index procesu v radku = pocet sloupcu pred nim
    if ( (is_local_col_0(ctx) || is_local_root(ctx)) && get_proc_line_ind(ctx) == solution_num ) {
        return true;
    }

    // V pripade, ze je proces v horni trojuhelnikove casti
    // procesni mrizky, tak plati :
    //
    // cislo_radku_v_procesni_mrizce = pocet_odpaleni_v_roli_COMMON
    if ( is_local_row_0(ctx) && solution_num == get_proc_line(ctx) ) {
        return true;
    }

    return false;
}
```

Výpis 13: Funkce change_role()

Po každém rozhodnutí se token s maticí přesune na transitní stav. Transitní stavy pro tento účel jsou dva, z jednoho se pak rozesílají tokeny na jiné *Compute* přechody, tj. dochází ke změně rolí (viz obr. 20, str. 47), u druhého zůstávají procesy v roli Common (viz obr. 23, 50).

Proces může token přeposlat na nový *Compute* přechod jen tehdy, až mu přijdou na místa *Right trigger* a *Bottom trigger* (viz obr. 20, str. 47) kontrolní tokeny (v síti značené jako *rt* a *bt*) od procesů sousedících s ním z pravé a dolní strany. Tyto kontrolní tokeny se zasílají po každém rozhodnutí o změně role, tedy současně s přesunem tokenu obsahujícího matici na transitní místo. Tak je zaručeno, že proces nemůže poslat data jinému procesu, kde vlivem zpoždění ještě nedošla data z minulého cyklu řešení a který by tak začal počítat se špatnými hodnotami.

Měření času je řešeno pomocí nástroje *system_clock* z C++ třídy *chrono* [23]. Měření provádí jen proces 0, který na konci řešení přijímá všechny vyřešené bloky. Ten na začátku sekvenčního kódu umístěného v přechodu *Compute ROOT* (viz obr. 19, str. 46) uloží do tokenu *begin_time* hodnotu, která značí okamžik, kdy k zápisu došlo. Ta je datového typu *time_point* [24]. Vyhodnocení času běhu LU-rozkladu je provedeno v přechodu *Print solution* (viz obr. 22, str. 49). Čas se získá rozdílem „timepointů", který je převeden na mikrosekundy funkcí *duration_cast()* [25] a jejich počet je získán metodou *count()* [26].

4.2.3 Výstup

Výstupem je celé číslo - čas běhu operací pro získání horní trojúhelníkové matice v mikrosekundách.

4.2.4 Možné optimalizace

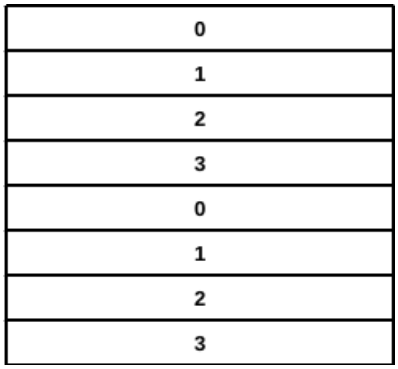
První věcí, která by se dala optimalizovat v implementaci tohoto programu je generování testovacích matic. V současné verzi programu jsou tyto vygenerovány mimo běh tohoto programu, což ztěžuje jeho použití. Nyní musí uživatel, ať už chce otestovat výkonnost programu ve srovnání s PETSc nebo pomocí něj testovat cluster, tento soubor s maticí získat a následně cestu k němu zadat při spouštění programu. Matice x-tisícího řádu zabírají řádově stovky megabytů, u řádů v desetitisících tato velikost rychle stoupá. Přenášení takových dat je již časově náročné a na disku osobního PC může velikost souboru „obtěžovat". Generování matice je dalším problémem, nedá se předpokládat, že pokud by program používal někdo jiný, bude disponovat také programem pro generování matic vhodných pro LU-rozklad.

Lepším řešením by tedy byla implementace generátoru matice přímo do programu - tak by se dal využít parametr *size*, resp. „velikost strany matice" (viz sekce 4.2.1, str. 31) pro vygenerování matice požadovaného řádu za běhu programu a uživatel by tak mohl používat program bez jakékoliv přípravy testovacích dat.

Další možností optimalizace je volba vhodnějšího algoritmu pro distribuci dat mezi procesy. 2D statická distribuce (viz obr.3, str.8), má v kombinaci s Gaussovu eliminační metodou (viz sekce 3.1, str.12) několik nevýhod. První z nich je malé využití výpočetních uzlů - po provedení výpočtu na své části bloku a odeslání dat už se tento uzel (s výjimkou uzlu 0) nijak nezapojuje a je tedy nevyužitý až do konce běhu programu. Druhou nevýhodou je pak velké množství přeposílaných dat. Každý proces (mimo posledního procesu na hlavní diagonále) v každé iteraci řešení zasílá data procesům vpravo a dole od něj (z pohledu procesní mřížky).

Problém s malým využitím výpočetních uzlů se dá vyřešit dynamickou distribucí - každý uzel by tak měl přidělen „svůj" blok matice, který řeší, ale po jeho vyřešení by mu byl přidělen jiný blok. Uzly by tak, až do konce běhu programu, byly vytíženy a stejného výsledku jako nyní u statické distribuce by se dalo dosáhnout s podstatně menším počtem výpočetních uzlů.

Problém s častým přeposíláním dat se dá řešit např. 1D distribucí po řádcích, resp. blocích řádků. Nyní je jeden řádek distribuován mezi více procesů a pro násobení řádků



Obrázek 15: 1D dynamická distribuce matice

nebo jejich sčítání s jiným řádkem je tak nutno všechna data (viz sekce 2.3.1.2, str. 9) v každém kroku výpočtu zasílat procesům umístěným více vpravo v procesní mřížce. Pokud by ale blok řádků měl jen jeden proces, mohl by zaslat *Multiplicator* a *MultRow* níže před začátkem vlastního výpočtu a poté řešit svůj blok matice. Zasílání *LineLeaders* by tímto vymizelo úplně, protože by každý proces znal hodnotu na začátku právě řešeného řádku.

Mým návrhem pro optimalizaci by tedy byla dynamická 1D distribuce po řádcích. Příklad je uveden na obrázku 15 (str. 36).

5 Návrhy na další vývoj nástroje Kaira

V této sekci jsou uvedeny návrhy pro zlepšení nástroje Kaira z ohledem na „uživatelskou přívětivost“. Nejedná se tedy o funkční chyby, jejichž zjišťování nebylo cílem této práce, ale spíše o návrhy pro zpříjemnění práce s nástrojem Kaira z pohledu uživatele, který s ním nikdy dříve nepracoval a není tedy navyklý na jeho „mechanismy“.

- První funkcionalitou, která by se mohla implementovat, je podpora knihoven, které využívají MPI. Tak by bylo možné, mimo jiné, použít v kombinaci s nástrojem Kaira i PETSc (viz sekce 3.7, str. 25). Díky vysokoúrovňovému rozhraní PETSc by bylo možno snadno využívat „obvyklé“ matematické operace, jako jsou např. maticové rozklady, bez nutnosti vlastní implementace. Program v nástroji Kaira by se tak zestručnil a omezila by se možnost chyb.

- Dále by byla užitečná možnost hromadného mazání prvků a mazání pomocí klávesy Delete. V současnosti je tato funkcionalita přístupná jen pomocí menu vyvolaného pomocí pravé klávesy myši, což může někdy při vývoji zdržovat.

- Mým největším problémem při práci s nástrojem Kaira bylo udržování přehlednosti sítě. Sice je zabudována možnost mřížky, ta však není viditelná a tak bych uvítal možnost jejího zobrazení, mohlo by to přispět k lepšímu vizuálnímu návrhu sítě. Se vzhledem souvisí i můj další návrh - možnost „rovné čáry“. Při vytváření dlouhých hran se velmi těžko udělá čára přesně vodorovná nebo svislá. Zajímavým doplňkem by tedy byla i klávesa pro zafixování rovné čáry, při jejímž držení by se čára přichytila v přesně horizontální nebo vertikální poloze.

- Při práci s velkou sítí jsem dále narazil na problém orientace v ní. Práci s takovou sítí by velmi urychlilo, pokud by byla zabudována možnost „vycentrování“ schématu pomocí tlačítka nebo klávesové zkratky.

- Posledním návrhem je přidání objektu s popiskem do grafického nástroje. Při návrhu velkých sítí jsem si často nebyl jistý, co jsem chtěl kterou částí řešit, velmi bych tedy ocenil možnost grafického „komentáře“ reprezentovaného štítkem s textem.

6 Výsledky testů

V této sekci jsou popsány naměřené časy běhu obou programů. Oba programy byly srovnávány při počítání s Gramovou maticí řádu 300 na 1, 4, 9, 16 a 25-ti procesech. Toto srovnání je ilustrováno grafem na obrázku 16 (str. 39).

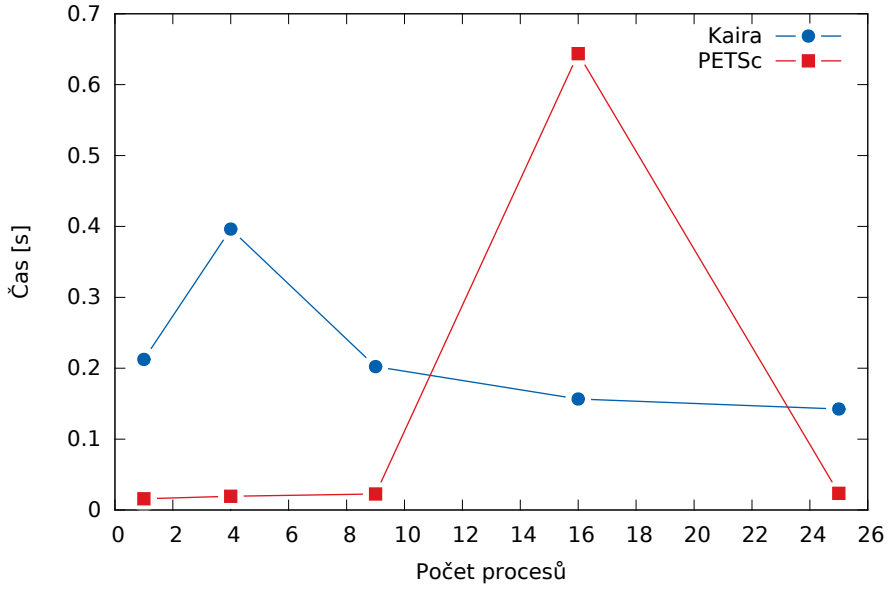
Větší rozdíly v naměřeném čase mezi počty procesů jsou vidět u programu vytvořeného v nástroji Kaira. U programu, který využívá PETSc jsou rozdíly v čase velmi malé. Výrazněji by se projevily u větších matic, které ale nebyly zvoleny z důvodu neoptimalizovaných funkcí PETSc pro práci s formátem MATELEMENTAL. Ty způsobovaly, že při spuštění programu na více než jednom procesu docházelo k velkému zpoždění při načítání, resp. konverzi matice (viz sekce 4.1.2, str. 28) a test tak nebyl vhodný pro účely této práce.

Pro ukázku výkonu samotného programu vytvořeného v nástroji Kaira byla měření provedena i na matici řádu 1800. Toto měření je znázorněno v grafu na obrázku 17 (str. 39).

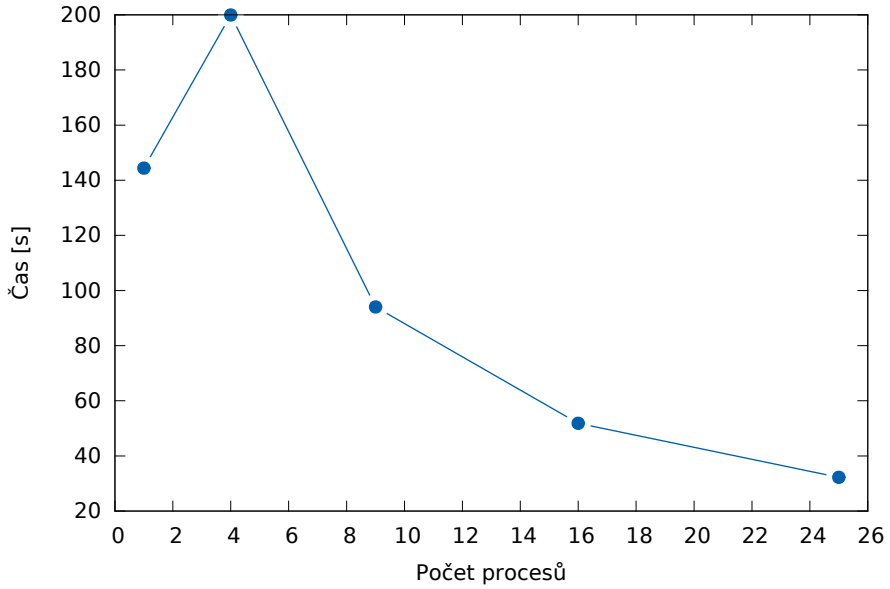
Všechny naměřené hodnoty jsou uvedené v tabulce 1.

		Technologie				
		PETSc		Kaira		
		300	1800	300	1800	**Řád matice**
Počet procesů	1	0.015805	0.4138	0.212540	144.410691	
	4	0.019338	-	0.396280	199.966221	
	9	0.022655	-	0.202227	94.009612	
	16	0.643750	-	0.156519	51.858760	
	25	0.023585	-	0.142416	32.224294	

Tabulka 1: Doby běhu LU-rozkladu v sekundách



Obrázek 16: Srovnání PETSc a Kairy



Obrázek 17: Kaira - měření na matici řádu 1800

7 Závěr

Výsledky testů ukázaly, že program vygenerovaný nástrojem Kaira je při počítání s velkými matice pomalejší než ten, který byl napsán sekvenčně s výpočtem prováděným pomocí funkcí knihoven PETSc. K tomuto výsledku však přispělo více faktorů, než jen samotná efektivita algoritmů pro generování sekvenčních programů v nástroji Kaira.

Hlavním z těchto faktorů byly jistě nedokonalosti v implementaci. Návrhy na jejich řešení jsou popsány v sekci *Možné optimalizace*. Vylepšení těchto nedostatků by udělalo testy více relevantními a výsledný program by pak mohl sloužit i jako vzor pro vývoj větších projektů v nástroji Kaira. Vzhledem k tomu, že je stále ve vývoji, v něm nebylo vytvořeno mnoho projektů a tato práce patří mezi nejrozsáhlejší.

Projekt Kaira sám o sobě je jistě perspektivní díky svému novému pohledu na vývoj paralelních systémů. Zcela odděluje sekvenční část a část paralelní, díky čemuž je vývoj mnohem přehlednější a pokud má vývojář s Kairou zkušenosti, pak jistě i rychlejší, než vývoj „čistým“ psaním zdrojového kódu. Tento přístup samozřejmě má i své nevýhody. Doslova nejviditelnější nevýhoda se projeví při implementaci algoritmu, který využívá často IPC a mnoho podmínek. Je to jev zřetelný i v této bakalářské práci. Schéma Petriho sítě je pak velmi rozsáhlé a pro člověka, který jej vidí jen zběžně, může být nepřehledné. Proto si myslím, že nástroj Kaira bude dále perspektivní pro specifické úlohy, kde se IPC tolik nevětví pomocí podmínek a spíše dochází k zasílání dat mezi procesy po „téže trase“.

Při tvorbě této práce jsem se dozvěděl mnoho informací o paralelních a distribuovaných systémech. Nabral jsem také praktické zkušenosti s vývojem paralelních systémů a získal kontakty na experty zabývající se tímto oborem. Tato bakalářská práce pro mě tedy byla velmi přínosná. Téma jsem si zvolil ze zvědavosti, bez toho, že bych měl nějaký přehled o tomto oboru, ale když jsem paralelní systémy poznal více, rád bych se jim nadále věnoval ve studiu i praxi. Zvláště mě zaujala problematika distribuce dat mezi procesy, jejíž efekt jsem viděl přímo v této práci, když jsem se zabýval popisem možných optimalizací. V další kariéře bych se chtěl tedy specializovat na tuto problematiku, zvláště při paralelizaci maticových rozkladů.

Za svůj největší přínos v této práci považuji „otestování“ nástroje Kaira při implementaci velkého projektu, nahlášení několika nově nalezených bugů a souhrn návrhů pro vylepšení Kairy, které jsem uvedl výše v popisu implementace. Myslím, že tyto návrhy budou prospěšné pro další vývoj Kairy, protože uživatelská zpětná vazba je zatím malá. Implementace některých těchto vylepšení pak umožní rychlejší a pohodlnější vývoj aplikací v tomto nástroji.

Martin Beseda

8 Reference

[1] *MPI Forum: MPI Documents*
URL: <http://www.mpi-forum.org/docs/docs.html> [cit. 2015-04-28]

[2] *Open MPI: FAQ Java*
URL: <https://www.open-mpi.org/faq/?category=java>
[cit. 2015-04-28]

[3] *MPI4Py - SciPy*
URL: <http://mpi4py.scipy.org> [cit. 2015-04-28]

[4] *MPICH: A High-Performance, Portable Implementation of MPI*, Argonne National Laboratory
URL: <http://www.mcs.anl.gov/project/mpich-high-performance-portable-implementation-mpi> [cit. 2015-04-28]

[5] *MPICH: MPICH Overview*
URL: <https://www.mpich.org/about/overview> [cit. 2015-04-28]

[6] *Open MPI: FAQ - General*
URL: <http://www.open-mpi.org/faq/?category=general#what>
[cit. 2015-04-28]

[7] *Open MPI: The Open MPI Development Team*
URL: <http://www.open-mpi.org/about/members> [cit. 2015-04-28]

[8] Blaise Barney: *MPI: Getting Started*, Lawrence Livermore National Laboratory
URL: <https://computing.llnl.gov/tutorials/mpi/#Getting_Started> [cit. 2015-04-28]

[9] Blaise Barney: *MPI: Point to Point Communication Routines*, Lawrence Livermore National Laboratory
URL: <https://computing.llnl.gov/tutorials/mpi/#Point_to_Point_Routines> [cit. 2015-04-28]

[10] Blaise Barney: *MPI: Blocking Message Passing Routines*, Lawrence Livermore National Laboratory
URL: <https://computing.llnl.gov/tutorials/mpi/#Blocking_Message_Passing_Routines> [cit. 2015-04-28]

[11] Blaise Barney: *MPI: Non-Blocking Message Passing Routines*, Lawrence Livermore National Laboratory
URL: <https://computing.llnl.gov/tutorials/mpi/#Non-Blocking_Message_Passing_Routines> [cit. 2015-04-28]

[12] Blaise Barney: *MPI: Collective Communication Routines*, Lawrence Livermore National Laboratory
URL: <https://computing.llnl.gov/tutorials/mpi/#Collective_Communication_Routines> [cit. 2015-04-28]

[13] *Kaira: About*
URL: <http://verif.cs.vsb.cz/kaira> [cit. 2015-04-28]

[14] *Kaira: Kaira User Guide*
URL: <http://verif.cs.vsb.cz/kaira/docs/userguide.html>
[cit. 2015-04-28]

[15] BÖHM, S. *Unifying Framework For Development of Message-Pass*. Faculty of Electrical Engineering and Computer Science VŠB – Technical University of Ostrava, 2013. [online] [cit. 2015-04-28] Dostupné z: http://verif.cs.vsb.cz/sb/thesis.pdf

[16] *GitHub: Kaira*
URL: <https://github.com/spirali/kaira> [cit. 2015-04-28]

[17] *Knihovna PETSc a cluster Hydra*, Ústav nových technologií a aplikované informatiky
URL: <http://www.nti.tul.cz/cz/Hydra/PETSc#Knihovna_PETSc_a_cluster_Hydra> [cit. 2015-04-28]

[18] *PETSc Component Overview*, Open Channel Foundation
URL: <http://www.openchannelfoundation.org/project/view_user_defined_page.php?user_defined_page_id=34&group_id=3>
[cit. 2015-04-28]

[19] *Kaira: Kaira User Guide: token_name*
URL: <http://verif.cs.vsb.cz/kaira/docs/userguide.html#_token_name> [cit. 2015-04-28]

[20] *Kaira: Kaira User Guide: pack & unpack*
URL: <http://verif.cs.vsb.cz/kaira/docs/userguide.html#_pack_amp_unpack> [cit. 2015-04-28]

[21] TURING, A. M. ROUNDING-OFF ERRORS IN MATRIX PROCESSES. *The Quarterly Journal of Mechanics and Applied Mathematics*. 1948, vol. 1, issue 1, s. 287-308. DOI: 10.1093/qjmam/1.1.287.

[22] GOLUB, Gene H. a Charles F. VAN LOAN. *Matrix computations*. 3rd ed. Baltimore: Johns Hopkins University Press, c1996, xxvii, 694 s. Johns Hopkins studies in the mathematical sciences. ISBN 08-018-5414-8.

[23] *C++ : Reference: <chrono>*
URL: <http://www.cplusplus.com/reference/chrono/> [cit. 2015-04-28]

[24] *C++ : Reference: time_point* URL: <http://www.cplusplus.com/reference/chrono/time_point/> [cit. 2015-04-28]

[25] *C++ : Reference: duration_cast*
URL: <http://www.cplusplus.com/reference/chrono/duration_cast/> [cit. 2015-04-28]

[26] *C++ : Reference: count*
URL: <http://www.cplusplus.com/reference/chrono/duration/count/> [cit. 2015-04-28]

[27] DORDA, M. *Nekonvenční metody 1* [online]. [cit. 2015-04-28] Dostupné z: http://homel.vsb.cz/~dor028/Nekonvencni_metody_1.pdf

[28] *A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, Innovative Computing Laboratory*
URL: <http://www.netlib.org/benchmark/hpl/> [cit. 2015-04-28]

[29] *A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary: HPL Algorithm, Innovative Computing Laboratory*
URL: <http://www.netlib.org/benchmark/hpl/algorithm.html>
[cit. 2015-04-28]

[30] SAPHIR, W., R. Van der WIJNGAART, A. WOO a M. YARROW. *Implementations and results for the NAS Parallel Benchmarks 2* [online]. 1994 [cit. 2015-04-28] Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/1994/npb_2.2.pdf

[31] WONG, P. a R. Van der WIJNGAART. *NAS Parallel Benchmarks I/O Version 2.4* [online]. 2003 [cit. 2015-04-28] Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-002.pdf

[32] JIN, H., M. FRUMKIN a J. YAN. T*he OpenMP Implementation of NAS Parallel Benchmarks and Its Performance* [online]. 1999 [cit. 2015-04-28]. Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf

[33] JIN, H., M. FRUMKIN a J. YAN. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance* [online]. 1999 [cit. 2015-04-28] Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf

[34] FRUMKIN, M., H. JIN a J. YAN. *Implementation of NAS Parallel Benchmarks in High Performance Fortran* [online]. 1999 [cit. 2015-04-28] Dostupné z: http://ipdps.cc.gatech.edu/1999/papers/114.pdf

[35] JIN, H. a R.F. Van der WIJNGAART. *NAS Parallel Benchmarks, Multi-Zone Versions* [online]. 2003 [cit. 2015-04-28] Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-010.pdf

[36] *Problem Sizes and Parameters in NAS Parallel Benchmarks, NASA Advanced Supercomputing Division*
URL: <http://www.nas.nasa.gov/publications/npb_problem_sizes.html> [cit. 2015-04-28]

[37] *NAS Software Select, NASA Advanced Supercomputing Division*
URL: <https://www.nas.nasa.gov/cgi-bin/software/start>
[cit. 2015-04-28]

[38] AALST, Wil van der a Kees van HEE. *Workflow management: models, methods and systems*. paperback ed. Cambridge: MIT Press, 2004, xvi, 368 s. ISBN 02-627-2046-9.

[39] *PETSc: Krylov Methods - KSP, Argonne National Laboratory*
URL: <http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/KSP/index.html> [cit. 2015-04-28]

[40] *PETSc: Preconditioners - PC, Argonne National Laboratory*
URL: <http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/PC/index.html> [cit. 2015-04-28]

[41] *PETSc: PCLU, Argonne National Laboratory*
URL: <http://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/PC/PCLU.html> [cit. 2015-04-28]

[42] *S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, K. Rupp, B. Smith, and H. Zhang: PETSc Users Manual, Argonne National Laboratory*
URL: <http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf> [cit. 2015-04-28]

[43] POSPÍŠIL, L. a V. VONDRÁK. *Numerické metody I* [online]. 2011 [cit. 2015-04-28] Dostupné z: http://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/numericke_metody.pdf

[44] *C++ : Reference: fstream*
URL: <http://www.cplusplus.com/reference/fstream/fstream/>
[cit. 2015-04-28]

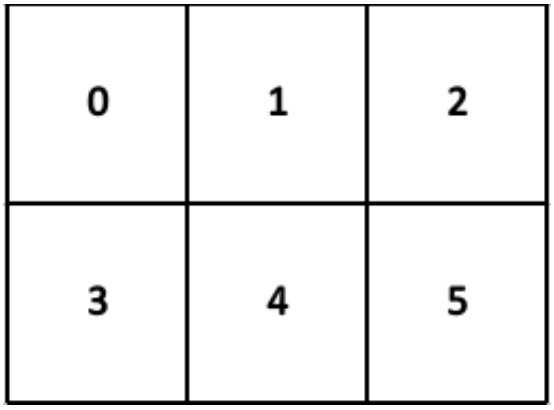
[45] *Armadillo*, National ICT Australia
URL: <http://arma.sourceforge.net/> [cit. 2015-04-28]

[46] *Armadillo: API Reference for Armadillo 5.100: Mat*, National ICT Australia
URL: <http://arma.sourceforge.net/docs.html#Mat> [cit. 2015-04-28]

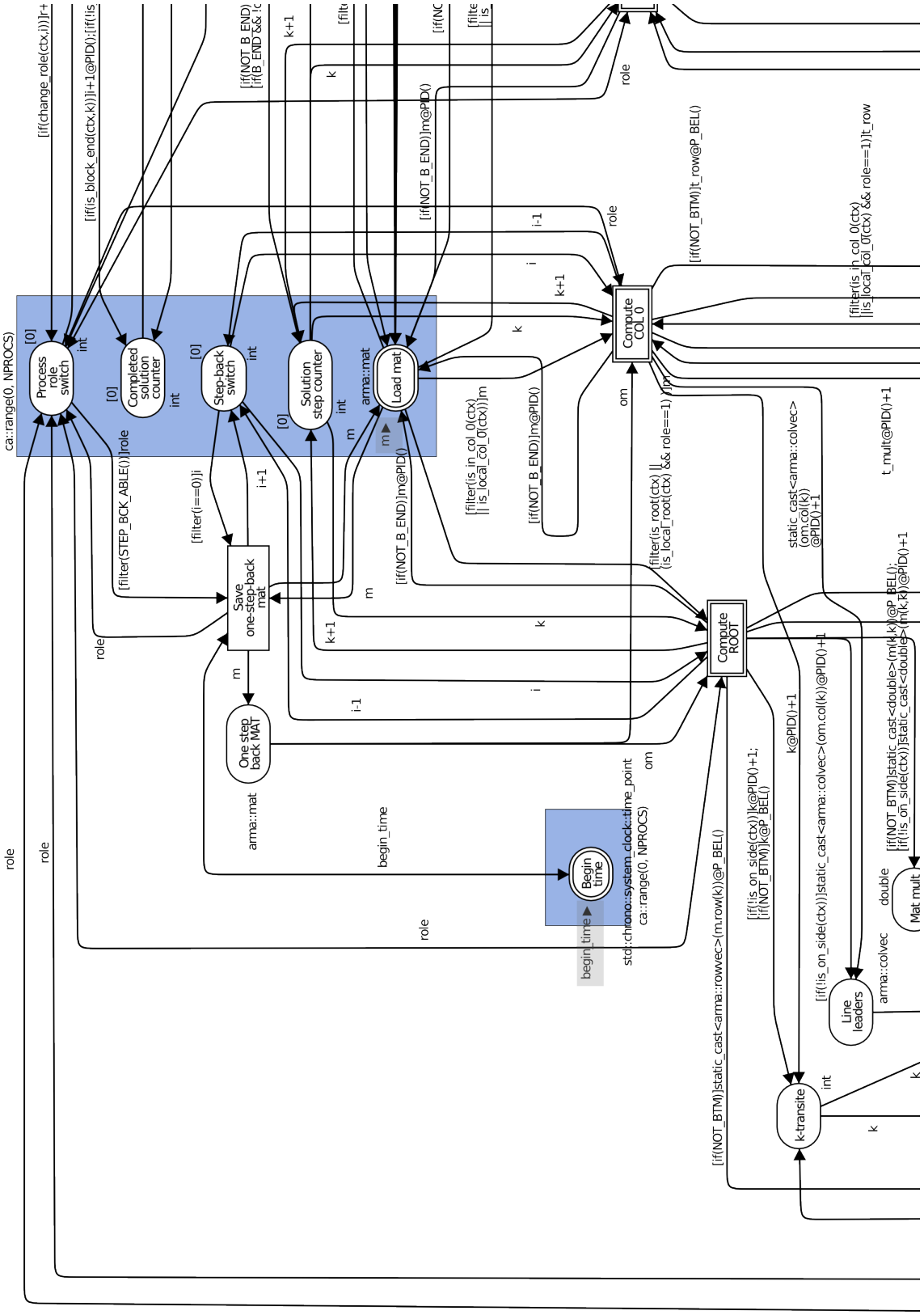
A Schéma v nástroji Kaira

Vzhledem ke značným rozměrům Petriho sítě vytvořené v nástroji Kaira a drobným popiskům, které by při zmenšení byly nečitelné, byla síť rozdělena na 6 částí v originální velikosti a vložena jako příloha. Každá část je na samostatné stránce a je očíslovaná v rozmezí 0-6. Rozložení těchto částí v celém schématu je na obrázku 18.

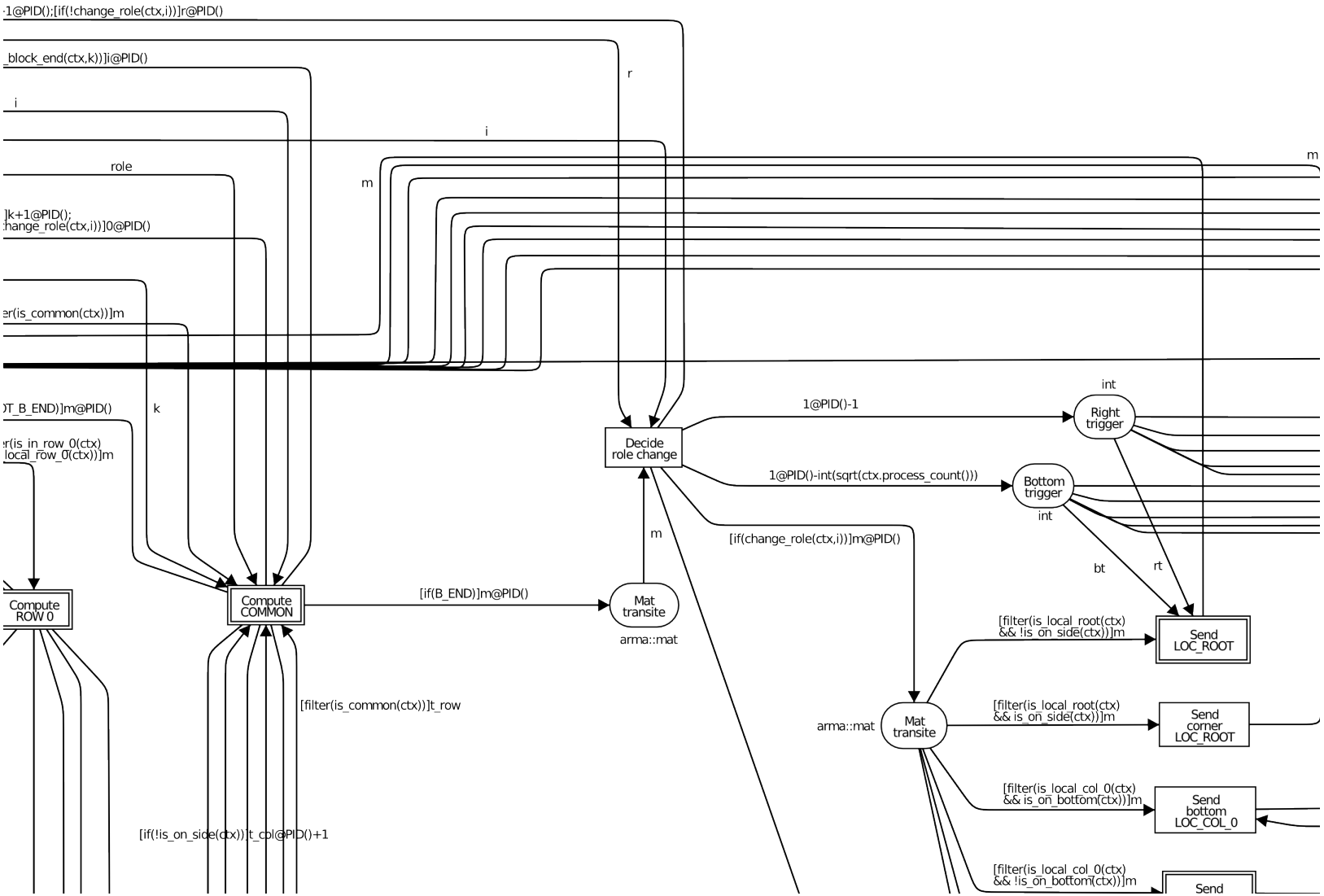
Celé schéma v originální podobě je v příloze na CD.



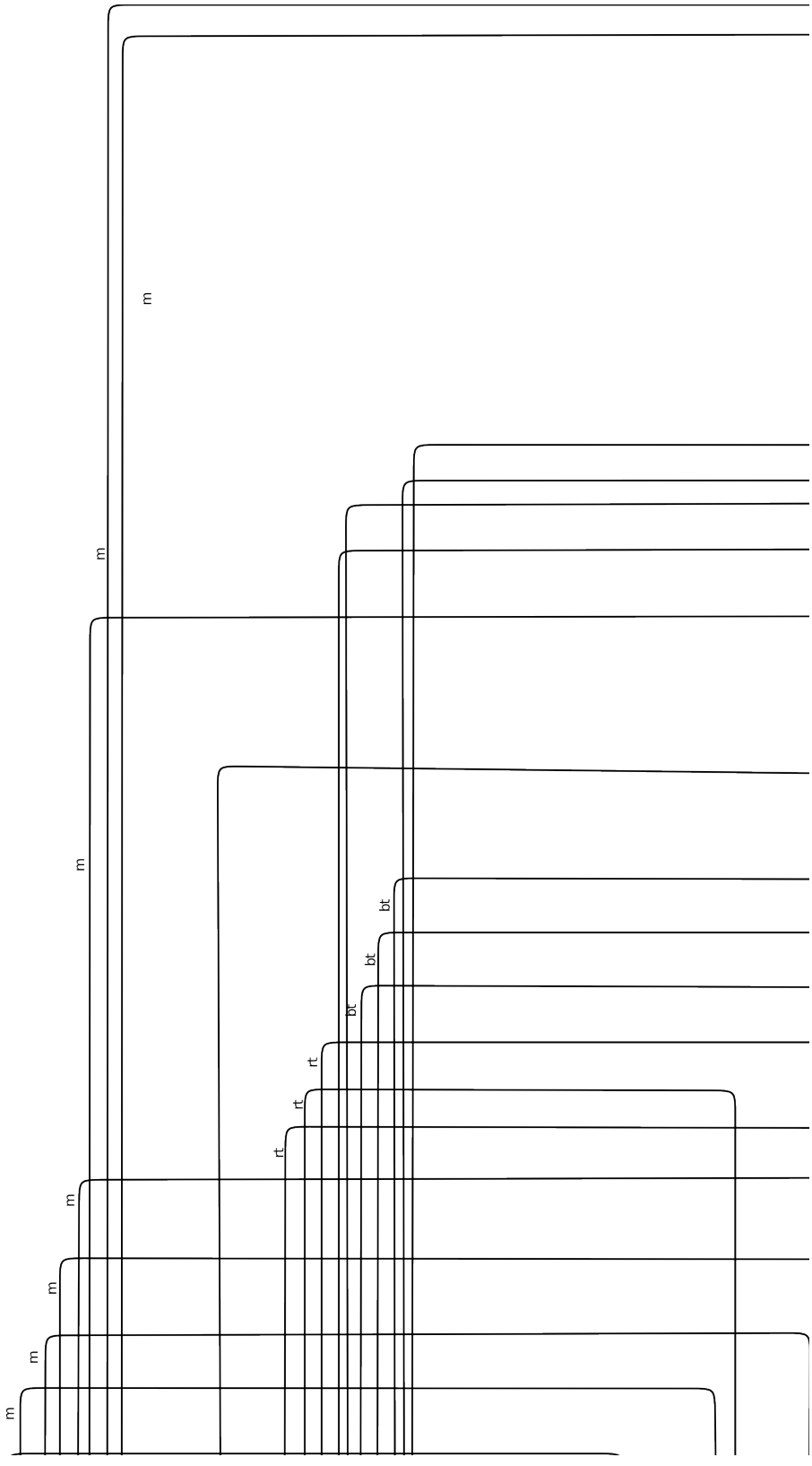
Obrázek 18: Rozložení částí sítě



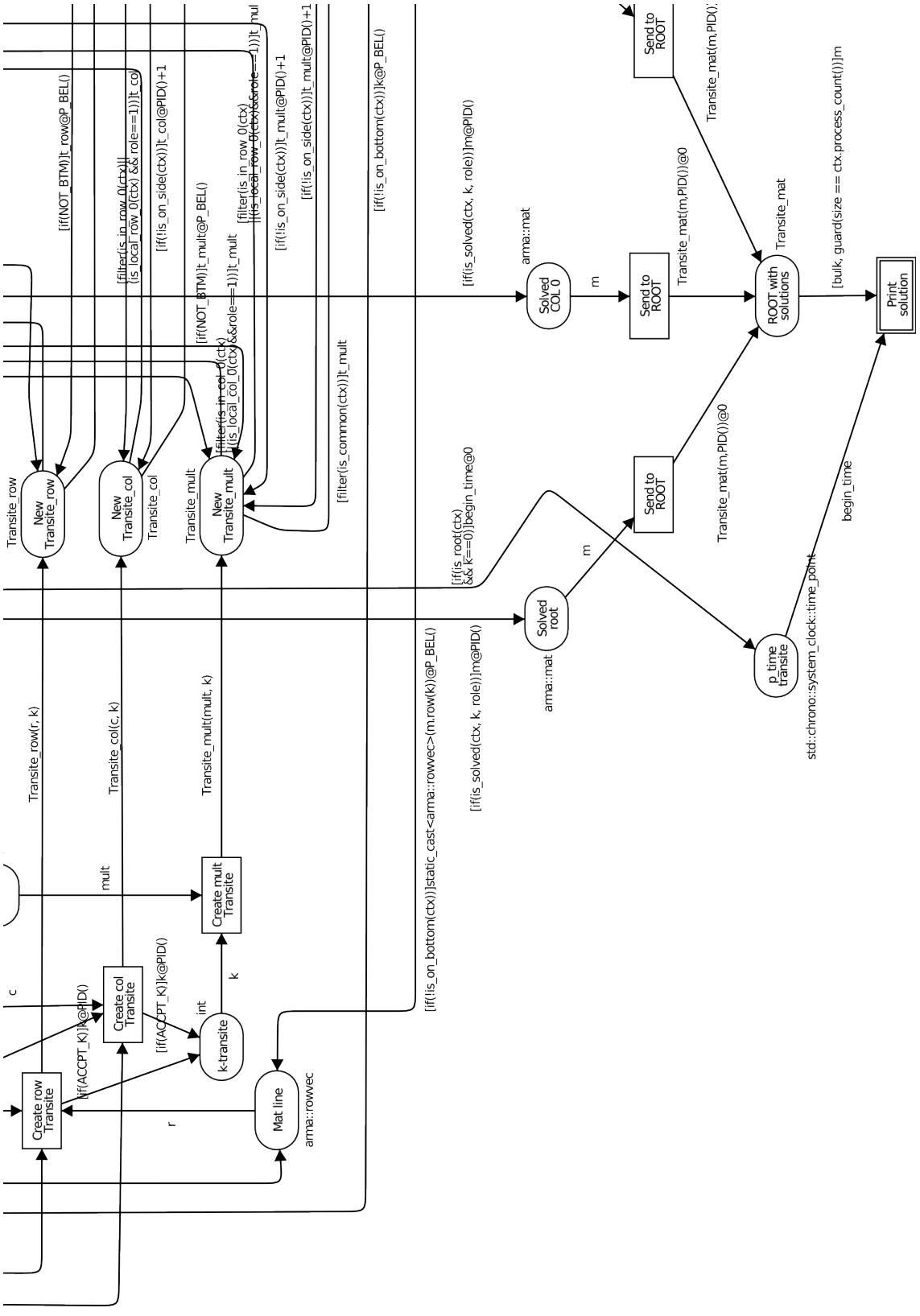
Obrázek 19: Síť v nátroji Kaira - část 0



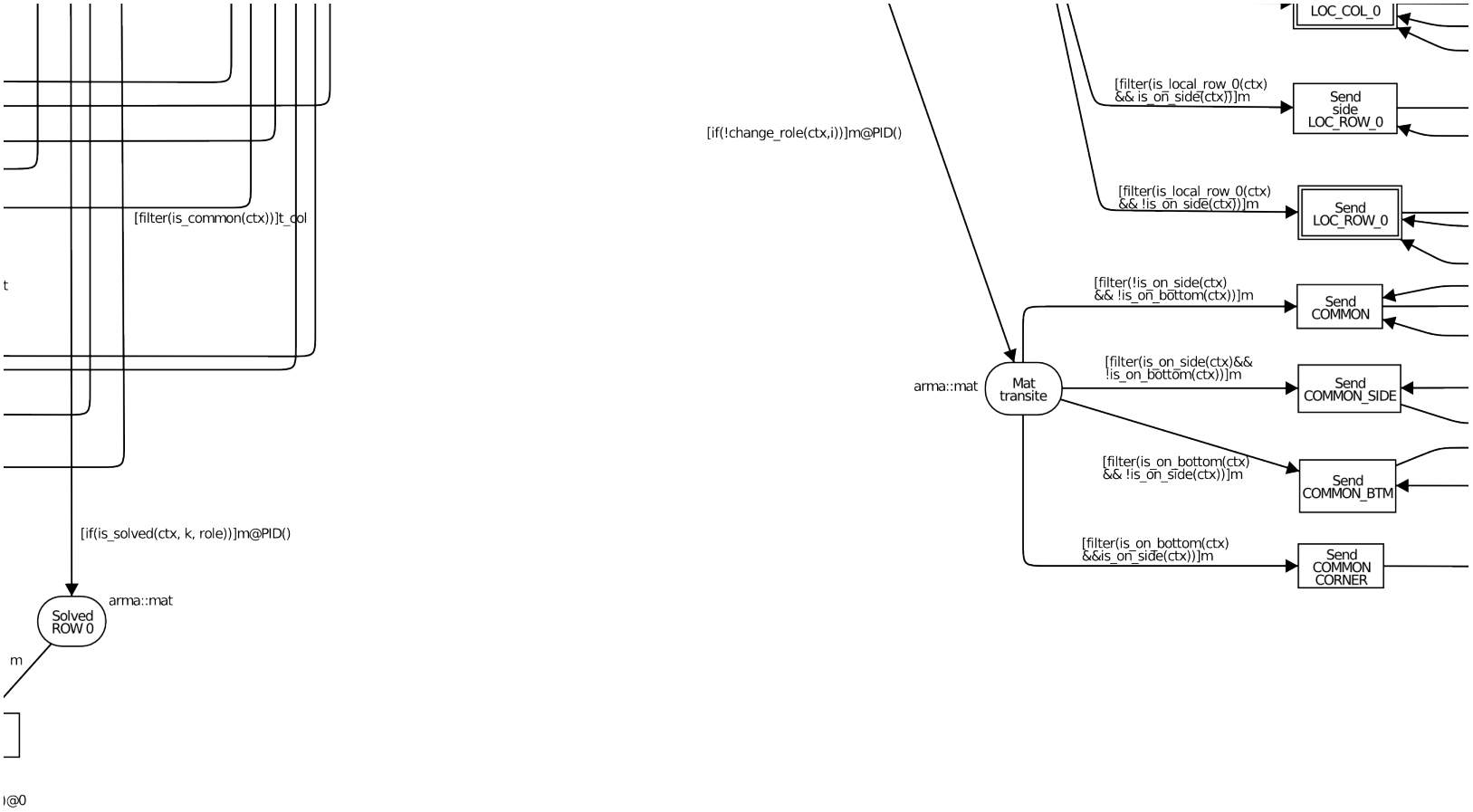
Obrázek 20: Síť v nátroji Kaira - část 1



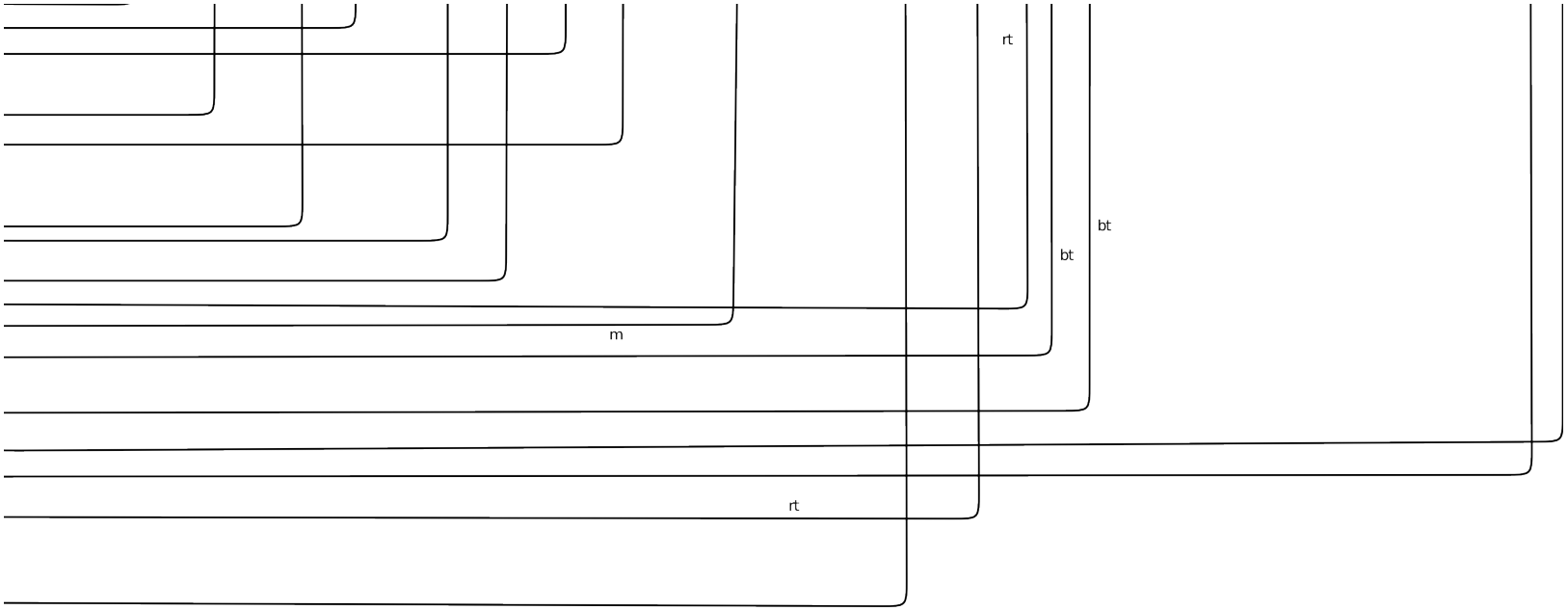
Obrázek 21: Síť v nátroji Kaira - část 2



Obrázek 22: Síť v nátroji Kaira - část 3



Obrázek 23: Síť v nátroji Kaira - část 4



Obrázek 24: Síť v nátroji Kaira - část 5

B Seznam příloh na CD

- Soubor s projektem nástroje Kaira
- Soubor se zdrojovým kódem programu využívajícího PETSc k řešení LU-rozkladu
- Soubor se zdrojovým kódem knihovny *matrixLib* s pomocnými funkcemi pro práci s maticemi
- Hlavičkový soubor knihovny *matrixLib*
- Soubor se zdrojovým kódem programu použitého ke generování Gramovy matice v binárním formátu
- Soubor se zdrojovým kódem programu použitého ke generování Gramovy matice v ASCII formátu
- Obrázek Petriho sítě navržené v nástroji Kaira
- Složka s nástrojem Kaira