Faculty of Informatics, Masaryk University

TCP vrstva pro verifikační nástroj DIVINE

Diplomová práce

Jiří Weiser

Brno, 2016

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: doc. RNDr. Jiří Barnat, Ph.D.

Abstrakt

V práci je prezentována motivace pro distribuované výpočty a jejich vztah ke knihovně MPI. Dále je v práci navržen a popsán nový přístup k řízení distribuovaného výpočtu, který lze použít v lokálních sítích. V závěru práce je nové řešení experimentálně porovnáno s knihovnou MPI. Porovnání je zaměřeno převážně na paralelní přístup ke komunikačnímu rozhraní.

Klíčová slova

Distribuované výpočty, MPI, Síť, TCP, Formální verifikace, Paralelizmus, DIVINE

Poděkování

Chtěl bych poděkovat svému vedoucímu docentu RNDr. Jiřímu Barnatovi, Ph.D., především za jeho trpělivost a ochotu mi poradit. Dále pak mé díky patří mým přátelům z laboratoře ParaDiSe, za příjemně strávené roky bádáním a diskusemi. V neposlední řadě bych chtěl poděkovat všem, kteří se kolem mne vyskytovali v době, kdy jsem psal tuto práci a měli pro mne pochopení.

Obsah

1 Úvod **1**
- 1.1 Graf . . . 2
- 1.2 Paralelizmus . . . 2
- 1.3 Formální verifikace . . . 4
- 1.4 DIVINE . . . 5
- 1.5 Cíl práce . . . 7

2 Komunikační vrstva **9**
- 2.1 MPI . . . 9
  - 2.1.1 Princip používání MPI . . . 10
  - 2.1.2 Koncepty MPI . . . 10
- 2.2 BSD sockety . . . 14
  - 2.2.1 Komunikační doména . . . 14
  - 2.2.2 Typy socketů . . . 15
- 2.3 Asio . . . 21
  - 2.3.1 Boost . . . 21
  - 2.3.2 Knihovna Asio . . . 22
  - 2.3.3 Princip používání Asio . . . 22
- 2.4 Stávající komunikační rozhraní . . . 23

3 Nové rozhraní **25**
- 3.1 Analýza vlastností typů socketů . . . 25
  - 3.1.1 Spojité sockety . . . 26
  - 3.1.2 Nespojité sockety . . . 28
  - 3.1.3 Sekvenční sockety . . . 29
  - 3.1.4 Zvolené řešení . . . 30
- 3.2 Architektura . . . 31
  - 3.2.1 Server . . . 31
  - 3.2.2 Klient . . . 32
- 3.3 Protokol . . . 33
  - 3.3.1 Formát zpráv . . . 33
  - 3.3.2 Seznam příkazů . . . 34
  - 3.3.3 Stavy serveru . . . 36

3.3.4 Ustanovení sítě . . . 37
3.3.5 Rozpuštění sítě . . . 40
3.3.6 Ostatní příkazy . . . 40
3.3.7 Řešení chyb a problémových stavů . . . 41
3.3.8 Bezpečnost . . . 42
3.4 Rozhraní pro distribuované procházení grafu . . . 42
3.4.1 Zprávy . . . 43
3.4.2 Služby komunikačního rozhraní . . . 43

4 Experimentální porovnání **47**
4.1 Test latence . . . 48
4.1.1 Porovnání dle scénářů . . . 48
4.1.2 Porovnání dle počtu vláken . . . 49
4.2 Testy škálovatelnosti . . . 50
4.2.1 Porovnání dle scénářů . . . 50
4.2.2 Porovnání dle počtu vláken . . . 52
4.3 Vyhodnocení . . . 55

5 Závěr **57**

A Struktura archivu a demonstrační program **59**
A.1 Serverové přepínače . . . 59
A.2 Klientská část . . . 60
A.2.1 Příkazy . . . 60
A.2.2 Výpočty . . . 60
A.2.3 Přepínače . . . 61

Literatura **61**

Kapitola 1

Úvod

Vývoj software se dlouhé roky zaměřoval převážně na sekvenční programy, které byly objevováním efektivnějších algoritmů, zaváděním pokročilých optimalizací a zrychlováním výpočetního hardware posunovány kupředu. V posledních několika letech došlo k situaci, že se výpočetní výkon přibližuje fyzikálním limitům, které se jen těžko budou překonávat. Pozornost se tedy začala ubírat směrem k paralelním programům.

Při vytváření programů dochází k chybám, proto byly vynalezeny různé postupy, které mají eliminovat množství chyb v programech. Mezi tyto postupy patří například zavedení typového systému, který některé chyby odhalí ještě před spuštěním programu, nebo testování, které se stalo nedílnou součástí vývoje software. Existují různé techniky testování, od jednotkového testování, pomocí kterého se hledají chyby v malých částech zdrojového kódu, až po integrační testy, pomocí nichž se hledají chyby ve výsledém programu. Na co ale nejde s úspěchem testování použít, jsou právě paralelní programy.

Problém s testováním paralelních programů je ten, že nikde není garantováno, v jakém pořadí se projeví výsledky souběžně vykonávaných příkazů. Nedeterminismus paralelních programů je tak veliký, že se mohou v programech nacházet závažné chyby, které mohou ohrožovat jak bezpečnost jeho uživatelů, tak samotný běh programu, které se navíc v rámci testování ještě neprojevily. Informatika se tomuto problému postavila zaváděním formálního ověřování paralelních algoritmů.

Používané metody formálního ověřování bývají náročné na výkon i na paměť. Výkonnostní problém se zpravidla řeší paralelizací samotného ověřování. Nároky na paměť je do určitého okamžiku možné řešit navyšováním pamětí v rámci jednoho stroje, nicméně i to má své limity, které se následně řeší zavedením distribuovaného výpočtu, kdy je paměť výpočtu fyzicky rozdělena mezi více výpočetních strojů.

Tématem této diplomové práce je prozkoumat různé přístupy k řízení distribuovaného výpočtu v kontextu verifikačního nástroje DIVINE. V následujících podkapitolách jsou postupně vysvětleny některé pojmy a v závěrečné

podkapitole je poté popsán cíl práce.

1.1 Graf

Pojem graf, se kterým dále operuji, znamená množinu objektů – dále označované jako vrcholy – jejichž vztahy jsou vyjádřeny hranami mezi nimi. Velmi neformálně a zjednodušeně je možné si graf představit jako mapu, kde města a vesnice představují vrcholy grafu a silniční síť mezi obcemi představují hrany. Nekdy je potřeba, aby hrany měly směr – tedy byly orientované – asi jako když je některá ulice jednosměrná. Formálnější definice (neorientovaného) grafu by mohla vypadat nějak takto:

> Neorientovaný graf je dvojce $G = (V, E)$, kde V je neprázdná množina vrcholů a E je množina hran, pro kterou platí $E \subseteq \{\{u, v\} \,|u, v \in V, u \neq v\}$.

Orientovaný graf se liší tím, že hrany mají směr a že je možné graf procházet po hranách jenom ve správném směru. Definice se liší jen v popisu množiny E:

> Orientovaný graf je dvojce $G = (V, E)$, kde V je neprázdná množina vrcholů a E je množina hran, pro kterou platí $E \subseteq V \times V$.

1.2 Paralelizmus

Pojem paralelizmus lze přeložit jako souběžnost. V informatice tento pojem znamená, že nějaká aplikace provádí (zdánlivě) současně dva nebo více nezávislých výpočtů. Paralelizmus můžeme dále rozdělit na dva druhy – paralelizmus ve sdílené paměti a v distribuované paměti. V případě paralelizmu ve sdílené paměti hovoříme o vlákně jako o základním funkčním prvku. U paralelizmu v distribuované paměti je základním prvkem proces.

Ačkoliv téměř každý člověk, který se pohybuje v oblasti informatiky, intuitivně tuší, co znamenají pojmy proces a vlákno, je obvykle obtížné uvést zcela přesnou definici. Definici, kterou zde uvádím, považuji za postačující pro potřeby této práce.

> Proces je instance počítačového programu, který je právě vykonáván. Proces si žádá od operačního systému zdroje jako například paměť. Proces nemá přímou možnost komunikovat s dalšími procesy a pro komunikaci je třeba využít služeb operačního systému.
>
> Vlákno je nejmenší sekvence příkazů, které mohou být nezávisle spravovány plánovačem úloh operačního systému. Vlákno je vždy součástí procesu a jako takové nevlastní žádné zdroje; ty náleží

procesu. Protože spolu vlákna sdílí paměť, komunikace mezi nimi probíhá bez přímé interakce s operačním systémem za použití synchronizačních primitiv dostupných na dané architektuře.

Důvod, proč není dostupná formální definice procesu, je ten, že pojem proces uvedli v 60. letech návrháři systému Multics [15] jako něco víc obecného než úkol, to v kontextu víceprogramové jednotky. Význam slova proces tak byl určen spíše implementací systému než zavedením formálního popisu, u čehož v současné době, kdy existuje několik operačních systémů s různou filozofií [20, 47], zůstalo . Obdobná situace je v případě definice pojmu vlákno, třebaže lze dohledat uvedení pojmu [37]. Nicméně zde platí stejně jako u procesu, že přesný popis, co vlákno je, se liší na základě operačního systému, případně běhovém prostředí[1]

V případě paralelizmu ve sdílené paměti programátor očekává, že po startu programu má k dispozici jedno hlavní vlákno, ze kterého následně v případě potřeby pouští další vlákna, na která většinou na konci programu hlavní vlákno zase počká. V průběhu výpočtu používají vlákna synchronizační primitiva pro vzájemnou komunikaci. Navíc platí, že programátor má volnou ruku v tom, zda spustí jednu a tu samou funkcí vícekrát ve více vláknech nebo spustí v některých vláknech jiné funkce.

V případě distribuovaného výpočtu je situace zcela opačná. Jeden program je s pomocí služeb operačního systému nebo nějaké knihovny spuštěn na různých výpočetních strojích, přičemž počet strojů – a tedy počet procesů – se během algoritmu zpravidla nemění. V průběhu výpočtu spolu mohou procesy komunikovat pomocí rozhraní, které jim poskytují podpůrné nástroje. Většinou je komunikace vedena formou zasílání zpráv. Je možné, že operační systém nebo knihovna umožňuje mezi procesy sdílet i části paměti, z důvodů výrazného zpomalení a problematické synchronizace ale nebývá její použití časté.

Z předchozích odstavců vyplývá, že přístup obou druhů paralelizmu je odlišný. Nejedná se tedy o konkurující, ale naopak doplňující se nástroje, které je možné využít při návrhu aplikace. Liší se i nástroje, kterými se oba druhy paralelizmu řídí. Pro paralelizmus ve sdílené paměti se používají nástroje jako POSIX vlákna [32] nebo OpenMP [23]. Paralelizmus v distribuované paměti zase využívají knihovny jako MPI [27] nebo PVM [28]. Není proto nezvyklé, že jsou v rámci jedné aplikace použity oba druhy paralelizace.

[1] Běhové prostředí je sada knihoven a nástrojů, které bývají součástí vyšších programovacích jazyků jako třeba Java nebo C#. Běhové prostředí může definovat jinou sémantiku především pro vlákna, které může být rozdílné od sémantiky, kterou popisuje operační systém.

1.3 Formální verifikace

Ve světě počítačů a programů se pojem formální verifikace užívá ve spojitosti se zjištěním, jestli nějaký program obsahuje chybu, či nikoliv. Přesnější definice je, že pomocí metod formální verifikace se dokazuje či vyvrací správnost systému. Obvykle nelze ověřovat správnost systému obecně, takže ověření bývá vztaženo na nějakou vlastnost systému, která je daná nějakým formálním popisem.

Existuje několik konkrétních metod, které spadají pod formální verifikaci, jako třeba theorem proving [29] nebo abstraktní interpretace [21, 22]. Pro nás je ale podstatný model checking [18], neboli také ověřování modelů. Metoda ověřování modelů je vhodným nástrojem na nalézání chyb obzvláště na paralelní programy, u nichž se případná chyba nemusí při běžném testování vůbec objevit.

Pro použití formální metody ověřování modelů je potřeba vzít model systému, který se vytvoří z popisu programu, který chceme ověřit. Další věc, kterou je potřeba mít, je specifikace vlastností, která vznikne převedením požadavků na systém do nějakého formálního jazyka. Obě vzniklé věci – model systému a specifikace vlastností – jsou vstupem ověřovacímu algoritmu. Pokud model systému splňuje specifikované vlastnosti, je ověřovacím algoritmem označen jako správný. Pokud ale model nesplňuje specifikaci, označí ho ověřovací algoritmus jako nesprávný a podá uživateli protipříklad, což je příklad chování modelu systému, které vede k porušení některé ze specifikovaných vlastností.

Algoritmus pro ověřování modelu je prováděn programem, kterému se říká *model checker*[2]. Každý *model checker* definuje, jakým způsobem má být zapsaný model systému a specifikace vlastností. Některé *model checkery* požadují, aby byl model systému napsán v nějakém speciálním modelovacím jazyku [6], jiné dokáží akceptovat jako model rovnou ověřovaný program ve formě zdrojových kódů. Mnohé *model checkery* dokáží přijímat více formátů vstupu. Stejná situace je u specifikace vlastností, kde ale záleží nejen na *model checkeru*, ale také na formátu modelu systému. Základní specifikační vlastnost je *assert* (tvrzení), kdy *model checker* ověří, že jsou všechna tvrzení, která nalezne, platná. Jiný možný formalizmus, který lze použít, jsou LTL [45] a CTL [19] formule, pomocí nichž je možné popisovat chování modelů, jejichž běh nikdy neskončí.

Ověřování modelů lze obecně provádět dvěma způsoby – symbolicky [41] a explicitně [19]. Symbolickému vyhodnocování se nebudu dále věnovat, pozornost věnuji naopak explicitnímu. Algoritmus pro explicitní ověřování modelů nejprve rozloží běh programu na jednotlivé stavy. V rámci každého stavu je uložen obraz paměti programu, včetně hodnot registrů[3] a ukazatele na ještě

[2] Seznam některých *model checkerů* je například na stránce `https://en.wikipedia.org/wiki/List_of_model_checking_tools`.

[3] Registry jsou pekelně rychlé malé paměti (řádově jednotky bytů), které procesor využívá

nespuštěnou instrukci. Stavy programu jsou vrcholy grafu, který se nazývá stavový prostor programu. Vrcholy grafu jsou spojeny orientovanou hranou, pokud se provedením jedné instrukce stane z jednoho vrcholu (předchůdce) druhý (následník). Pokud bude ověření podroben program bez paralelizmu, ve výsledném stavovém prostoru budou nejspíše všechny vrcholy pospojovány v řadě za sebou. Ovšem v okamžiku, kdy bude ověřovaný program používat paralelizmus, začne být situace zajímavější – pokaždé, kdy bude v programu více vláken, *model checker* musí postupně provést instrukci za každé jedno vlákno, takže z předchůdce vzniknou hrany do více následníků.

Další zpracování se liší typem použité specifikační vlastnosti. Pro nejjednodušší typ – *assert* – postačí při vytváření stavového prostory označit ty vrcholy, v nichž požadované tvrzení neplatí. *Model checker* poté projde graf, k čemuž využije převážně algoritmy na procházení grafu do hloubky a do šířky, a pokud nalezne nějaký označený vrchol, nahlásí model za chybný. V opačném případě je model korektní.

U obsáhlejších typů specifikačních vlastností je potřeba modifikovat graf a použít jiné algoritmy. Například v případě LTL formulí musí nejprve *model checker* z formule vytvořit její negaci, ze které následně vybuduje Büchi automat [13]. Výsledný automat se vynásobí se stavovým prostorem, a poznačí se všechny stavy, ve kterých platí negovaná formule. Pro zjištění, jestli je model systému správný, musí *model checker* použít sofistikovanější algoritmy prohledávání grafu, protože hledá cyklus, který obsahuje alespoň jeden poznačený cyklus. Model systému je platný, pokud se žádný takový cyklus ve stavovém prostoru nenachází.

Postup, který jsem v předchozích odstavcích předestřel, je pouze funkční základ. Reálné *model checkery* ho rozšiřují o mnohá vylepšení, jako třeba *partial order reduction* [44]. Stavový prostor dále bývá generovaný postupně, jak *model checker* prochází grafem a generuje následníky zpracovávaného vrcholu. Jedním takovým reálným *model checkerem* je nástroj DIVINE.

1.4 DIVINE

Nástroj DIVINE [11, 10] je explicitní *model checker*, který zvládá verifikovat modely v jazycích jako LLVM [39], UPPAAL [38][4], nebo ve formátu DVE[5]. Specifikace vlastností lze zadávat jako *asserty* nebo vyjádřit pomocí LTL formulí. V současné době je snaha soustředit se na jazyk LLVM a postupně rušit podporu pro ostatní vstupní formáty.

Proč tomu tak je? Jazyk LLVM je jednoduchým mezijazykem navrženým především pro snadnou implementaci optimalizací tak, aby zůstaly odstíněny od abstrakcí vyšších programovacích jazyků a zároveň aby nebyly omezo-

k výpočtům. Podle architektury jich procesor může mít jednotky [5] až malé desítky [1, 7].

[4] Formát pro vytváření časových automatů [9].

[5] Původní modelovací jazyk, ve kterém nástroj DIVINE uměl verifikovat modely.

vány prvky konkrétních architektur. První front-end, který překládal vyšší programovací jazyky do LLVM byl Clang, který překládá jazyky C, C++ a Objective-C. Posléze začaly vznikat front-endy i pro další jazyky, jako je Java, C#, Haskell a další. Pro velký úspěch byl celý projekt LLVM adoptován společností Apple Inc. Jazyk LLVM je v nástroji DIVINE upřednostňován pro svoji jednoduchost, protože je pak poměrně snadné generovat stavy programu, a zároveň existenci mnoha překladačů pro populární programovací jazyky, což může vést k použití nástroje DIVINE pro verifikaci reálných programů namísto upravovaných modelů systémů.

Pro generování stavového prostoru se používá interpret jazyka LLVM. To znamená, že nástroj DIVINE potřebuje pro verifikaci mít nejen zdrojové soubory ověřovaného programu, ale i zdrojové soubory všech knihoven, které program používá. Pro jazyky C a C++ jsme tento problém vyřešili tak, že spolu s nástrojem DIVINE distribuujeme standardní knihovny pro oba jazyky. Ostatní programovací jazyky proto nástroj DIVINE zatím nepodporuje.

Je potřeba si uvědomit, že graf stavového prostoru, který vzniknul interpretací mezijazyka LLVM má dvě podstatné vlastnosti. Velikost grafu narůstá s velikostí programu, a to jak kvůli délce programu, kde je nárůst lineární, tak kvůli počtu vláken ověřovaného programu, kde je nárůst exponenciální. Samotné vrcholy grafu také nejsou malé, neboť každý vrchol potřebuje obsáhnout celou paměť programu v určitém místě výpočtu. Nástroj DIVINE se se skutečností, že potřebuje pracovat s obrovskými grafy[6], které mají zároveň velké vrcholy[7], vypořádává několika způsoby. Zavádí bezztrátovou kompresi stavů [56], což snižuje velikosti jednotlivých vrcholů. Dále používá *partial order reduction* a spojování instrukcí[8], což vede ke zmenšování velikosti grafu.

Nadále velké grafy, třebaže zmenšené, je potřeba co nejrychleji zpracovat. Nástroj DIVINE proto využívá oba druhy paralelizace, přičemž v aktuální verzi[9] je upřednostňován režim paralelizace pouze ve sdílené paměti. Některé důvody jako například možnost komprese stavového prostoru nebo rovnoměrnější rozvržení pracovní zátěže jednotlivých vláken, což vede k rychlejšímu prohledávání stavového prostoru, jsou popsány v [56] a v mé bakalářské práci [57].

Druhý režim paralelizace je hybridní a zahrnuje oba dva druhy paralizace. Tento režim pochází ze starší verze programu[10] a oproti původní verzi nebyl nikterak vylepšován (až na malé optimalizační změny). Hybridní paralelizmus je realizován tak, že každý stav ze zpracovávaného stavového prostoru je sta-

[6] Pomocí nástroje DIVINE jsme verifikovali některé jeho komponenty, což mnohdy vyústilo v grafy o několika stovkách milionů vrcholů.

[7] Běžně dosahují několika jednotek nebo malých desítek kilobytů.

[8] Při interpretaci není potřeba po každé provedené instrukci vytvořit nový stav, pokud výsledek této instrukce není viditelný z jiného vlákna.

[9] DIVINE 3.x

[10] DIVINE 2.x

ticky přiřazen některému vláknu na některé samostatné výpočetní jednotce pomocí hašování [35].

Nevýhodou hybridního režimu paralelizace je statické rozdělení stavů nejen mezi jednotlivé výpočtní stroje, ale i mezi jednotlivá vlákna. Toto rozdělení má kromě nevýhody v potenciálně nerovnoměrném rozložení práce mezi jednotlivá vlákna i nevýhodu v nemožnosti použít aktuální implementaci komprese paměti bez velké paměťové režie.

V současné době se pracuje na nové verzi programu DIVINE, přičemž součástí změn je i úprava modelu paralelního zpracování stavového prostoru a zavedení jednotného režimu paralelizace pomocí dvouvrstvé architektury. Z tohoto důvodu jsme zvažovali, jestli by nebylo výhodné zaměnit stávající řešení řízení distribuovaného výpočtu za jiné.

1.5 Cíl práce

Zde konečně přichází objasnění zadání mé dimplomové práce. Mým cílem je navrhnout a implementovat novou komunikační vrstvu, kterou by nástroj DIVINE používal namísto stávájícího MPI standardu [27][11]. Součástí práce je i porovnání nového a stávajícího komunikačního rozhraní.

Motivací pro implementaci vlastní komunikační vrstvy je několik. Předně je zde očekávání, že komunikační vrstva psaná na míru nástroji DIVINE bude rychlejší než stávající řešení, které je obecné. Dále je předpoklad, že pokud bude komunikace postavena přímo nad síťovým protokolem, odpadne nutnost přikládat k nástroji DIVINE další knihovnu, což činí režii nám vývojářům a může činit potíže uživatelům nástroje. A naposled je motivací touha mé maličkosti vyzkoušet si práci se sítí na projektu, který má budoucnost.

[11] O standardu MPI je napsáno více v kapitole 2.

Kapitola 2

Komunikační vrstva

V této kapitole pojednávám o hlavních adeptech pro použití jako komunikační kanál v distribuované paralelní aplikaci. Ve čtvrté podkapitole je popsaná komunikační vrstva tak, jak je používaná v aktuální verzi nástroje DIVINE.

2.1 MPI

Message Passing Interface, zkráceně MPI, je standardizovaný systém pro distribuované výpočty, což zahrnuje zasílání zpráv mezi procesy a prvky kolektivní komunikace. Ačkoliv se většinou mluví o systému, MPI jako takové je standard[1] a pro použití je potřeba použít některou z implementací.

Samotný standard MPI se zmiňuje o rozhraní pouze pro jazyky C a Fortran. Ve 2. verzi standardu MPI byla přidána podpora pro jazyk C++, která byla hned ve verzi 3 odstraněna. Z základních implementací jmenujme alespoň MPICH[2], Open MPI, či LAM/MPI, jehož vývoj byl zastaven ve prospěch Open MPI. Pro jiné programovací jazyky existují moduly, které používají rozhraní pro jazyk C z dostupné implementace standardu MPI. Jde mimo jiné o programovací jazyky jako Java [16], Python[3], jazyk R [17], nebo o knihovnu dostupnou pro framework .NET [30].

MPI je navržen jako multiplatformní systém, což umožňuje z pohledu aplikace nezohledňovat detaily architektury, jako jsou například endianita[4] nebo rozsahy čísel. Kvůli tomu například zavádí MPI vlastní datové typy. Z pohledu výpočetních strojů nabízí MPI možnost dodat vlastní implementaci – tzv. poskytovatele – pro specifický hardware a dosáhnout tím lepšího výkonu distribuovaného výpočtu. Na běžných strojích v lokální síti MPI obvykle používá jako poskytovatele síťovou infrastrukturu, v případě, že je distribuovaný

[1] Ačkoliv má MPI standard již několik verzí, stále nedošlo ke standardizačnímu řízení u některé ze známých standardizačních autorit, jako je například ISO nebo IEEE.

[2] MPICH byl vůbec první implementací MPI standardu, konkrétně MPI-1.1.

[3] `http://sourceforge.net/projects/pympi/` a `https://code.google.com/p/pypar/`

[4] Způsob ukládání čísel do paměti.

výpočet spuštěn v rámci jednoho stroje, pak může využít možností operačního systému a mít poskytovatele pro sdílenou paměť.

2.1.1 Princip používání MPI

Jako framework pro podporu paralelizmu v distribuované paměti poskytuje MPI několik nástrojů. Jsou jimi knihovna a hlavičkové soubory, které exportují deklarace funkcí a definice struktur, vlastní překladač, který způsobí připojení knihoven MPI k programu, a speciální zaváděcí program, pomocí které lze distribuovaný výpočet spustit.

Jakýkoliv program, který má být spuštěn pomocí MPI jako distribuovaný výpočet, musí nejprve inicializovat MPI knihovnu voláním funkce `MPI_Init`. Následně je vhodné zjistit, co je každý proces zač, k čemuž slouží funkce `MPI_Comm_rank` a `MPI_Comm_size`. První z nich vrací rank procesu a druhá počet procesů. Rank je číslo, které udává pořadí procesu, je číslované od 0 až po $N - 1$, kdy N je počet procesů. Dále pokračuje běh výpočtu, který se obvykle řídí zasíláním zpráv. Před ukončením výpočtu musí každý proces zavolat funkci `MPI_Finalize`, která korektně ukončí spojení s ostatními procesy.

Pro spuštění programu je nejprve potřeba přeložit zdrojový kód a připojit k výslednému programu právě knihovnu MPI. To je možné udělat buď přeložením zdrojového kódu pomocí `mpicc` nebo `mpic++`, překladače dodaného implementací MPI standardu, nebo svému překladači nastavit vhodné systémové cesty právě ke knihovně MPI, aby ji mohl připojit k výslednému programu[5].

Následné spuštění distribuovaného algoritmu se provede spuštěním zaváděcí program `mpirun` nebo `mpiexec`[6], který bývá nazýván MPI agentem. V rámci spuštění je potřeba definovat některé parametry, z nichž nejpodstatnější je seznam strojů, na kterých mají běžet procesy. Další parametry pak bývají volitelné a je možné jimi nastavit mnoho vlastností běhu.

2.1.2 Koncepty MPI

MPI poskytuje různorodou schopnosti a nástroje pro řízení a běh distribuovaného výpočtu. Uvádím zde čtyři základní koncepty MPI, které byly uvedeny již v první verzi standardu. Ve verzi 2 přibylo dalších několik konceptů jako například sdílení paměti mezi procesy, dynamické vytváření procesů nebo podpora paralelních vstupně-výstupních operací. Protože ale patří tyto koncepty mezi pokročilé a obtížnější, nebudu je zde uvádět.

[5] Což je ostatně přesně to samé, co provede překladač dodaný implementací MPI.
[6] Jsou totožné.

Komunikátor

Komunikátor ustanovuje skupinu procesů. Po spuštění programu jsou všechny procesy součástí skupiny `MPI_COMM_WORLD`. V průběhu výpočtu mohou procesy zakládat a rušit další skupiny, což může vést k dynamickému vytváření topologie v komunikaci. Každý proces připojením ke komunikátoru dostane další rank, který určuje jeho pořadí v rámci skupiny.

Jeden na jednoho

Jde o důležitý mechanizmus, který umožňuje posílat zprávy od jednoho procesu ke druhému. Komunikace jeden na jednoho je vhodná zvláště v případech nahodilé komunikace, kdy nelze dopředu říct, který proces zašle data kterému, nebo v případě, že jsou procesy uskupeny do rolí pána a otroka – jeden proces řídí část nebo celý výpočet a jeden nebo více procesů přijímá příkazy a provádí výpočet. Zasílané zprávy je možné třídit pomocí štítku zprávy – příjemce může definovat, zda ho zajímají zprávy s nějakým konkrétním štítkem, nebo všechny zprávy.

MPI nabízí několik variant zasílání zpráv, které se liší především svým vztahem k paměti a k blokování výpočtu. Jsou dvě základní rozdělení. První dělí operace na blokující a neblokující, druhé je dělí na přímé (nebafrované) a na ty s vyrovnávací pamětí (bafrované).

`MPI_Send` – **blokující nebafrované**
: Funkce odešle zprávu a blokuje až do okamžiku, když přijímající proces započal příjem zprávy a zpráva byla úspěšně odeslána. Po skončení funkce jsou odeslaná data plně k dispozici.

`MPI_Bsend` – **blokující bafrované**
: Funkce nakopíruje zprávu do vyrovnávací paměti, zahájí přenos zprávy a skončí. Po dokončení přenosu je potřeba obsah vyrovnávací paměti zlikvidovat.

`MPI_Isend` – **neblokující nebafrované**
: Funkce zahájí přenos zprávy a skončí. Zprávu nesmí odesílatel modifikovat, dokud si neověřil, že bylo odeslání dokončeno. To je možné provést voláním funkcí `MPI_Test` nebo `MPI_Wait`.

`MPI_Ibsend` – **neblokující bafrované**
: Funkce zahájí kopírování zprávy do vyrovnávací paměti, zahájí přenos zprávy a skončí. Zprávu nesmí odesílatel modifikovat, dokud si neověřil, že bylo odeslání dokončeno. To je možné provést voláním funkcí `MPI_Test` nebo `MPI_Wait`.

Pro příjem zprávy slouží primárně dvě funkce, opět dělené na blokující a neblokující.

`MPI_Recv` – **blokující**
: Funkce čeká na zprávu, dokud není zpráva doručena. Příchozí zpráva musí mít správného odesílatele a správnou hodnotu štítku, aby byla přijata, ale je možné ignorovat jak rank odesílatele, tak hodnotu štítku zprávy. Je třeba dopředu nastavit dostatečně velkou paměť pro příjem zprávy.

`MPI_Irecv` – **neblokující**
: Funkce zahájí příjem zprávy, předá zpátky kontrolní strukturu a skončí. Příchozí zpráva musí mít správného odesílatele a správnou hodnotu štítku, aby byla přijata, ale je možné ignorovat jak rank odesílatele, tak hodnotu štítku zprávy. Příjem zprávy je potřeba ověřit voláním `MPI_Test` nebo `MPI_Wait` s kontrolní strukturou jako parametrem. Je třeba dopředu nastavit dostatečně velkou paměť pro příjem zprávy. Pokud je iniciován příjem zprávy, je nutné tuto zprávu přijmout před ukončením výpočtu.

Pokud není dopředu známé, zda vůbec nějaká zpráva dojde, případně není známá její velikost, nabízí se použít dvou funkcí na zjištění příchozí zprávy. Funkce jsou opět dělené na blokující a neblokující.

`MPI_Probe` – **blokující**
: Funkce čeká, dokud nepřijde nějaká zpráva. Příchozí zpráva musí mít správného odesílatele a správnou hodnotu štítku, aby byla přijata, ale je možné ignorovat jak rank odesílatele, tak hodnotu štítku zprávy. Poté vrátí vlastnosti zprávy – rank odesílajícího procesu, štítek zprávy a velikost zprávy v bytech.

`MPI_Iprobe` – **neblokující**
: Funkce zjistí, zda na příjem nečeká nějaká zpráva od správného odesílatele a se správnou hodnotu štítku. Je možné ignorovat jak rank odesílatele, tak hodnotu štítku zprávy. Pokud funkce zjistí příchozí zprávu, je možné zjistit její vlastnosti – rank odesílajícího procesu, štítek zprávy a velikost zprávy v bytech.

Kolektivní komunikace

Kolektivní komunikace znamená, že všechny procesy začnou společně provádět nějakou operaci nad daty. Počet procesů je možné omezit vytvořením nového komunikátoru a provedením operace v něm. Ačkoliv je možné všechny kolektivní operace simulovat pomocí operací jeden na jednoho, je výhodnější použít přímo funkce pro kolektivní komunikaci, protože jejich použitím může MPI knihovna využít znalosti o topologii procesů a realizovat přenos dat efektivněji.

Mezi základní kolektivní operace patří jeden-všem, kdy jeden proces prošle stejná data všem ostatním procesům, všichni-jednomu, kdy všechny procesy

až na jeden zašlou data jednomu procesu, přičemž data jsou před přijetím podrobeny redukční operaci, a bariéra, což je synchronizační primitivum, kterého musí všechny procesy ve skupině dosáhnout, než jim všem bude umožněno pokračovat dále ve výpočtu. Obdobně jako u komunikace jeden na jednoho i operace kolektivní komunikace mají varianty v podobě neblokujících volání; ověření dokončení operací se řeší také stejně.

`MPI_Bcast`
: Funkce realizující broadcast – rozeslání balíku dat všem ostatním procesům ve skupině.

`MPI_Reduce`
: Funkce realizující redukci dat – pošle data jednomu procesu od všech ostatních ze skupiny. Pro redukční operaci je možné využít buď jednu z předdefinovaných operací jako je součet, maximum, či logický součet, nebo si může uživatel definovat pomoc funkce `MPI_Op_create` vlastní operaci.

`MPI_Barrier`
: Funkce realizující bariéru.

Pokročilější kolektivní operace zahrnují rozesílání a sbírání různých dat od procesů ve skupině. Další operace jako `MPI_Allgather` nebo `MPI_Allreduce` kombinují více kolektivních operací do jedné.

`MPI_Scatter`
: Funkce rozešle balíky dat všem procesům ve skupině. Rozdíl oproti funkci `MPI_Bcast` je v tom, že odesílající proces sestaví různá data pro každý přijímající proces.

`MPI_Gather`
: Funkce přijme data od všech ostatních procesů ve skupině. Je podobná funkci `MPI_Reduce` s rozdílem, že není definována redukční operace, ale přijímající proces má k dispozici všechna data.

Vlastní datové typy

Standard MPI zavádí vlastní datové typy, které jsou schopné reprezentovat většinu základních datových typů v jazycích C, C++ a Fortran. Naopak některé MPI datové typy nemají reprezentaci ve jmenovaných programovacích jazycích. Jedním z důvodů zavedení vlastních datových typů je nezávislost na architektuře – především na endianitě. Dalším důvodem je možnost používat předdefinované operace pro kolektivní komunikaci.

Kromě vlastních datových typů umožňuje MPI definovat i vlastní datové typy, což zahrnuje vytváření jak homogenních (pole), tak i heterogenních (struktury) datových typů. Spolu s uživatelsky definovanými operacemi

je možné využít vlastní datové typy při redukcích během kolektivní komunikace.

2.2 BSD sockety

BSD sockety je rozhraní pro přístup k síťové komunikaci a k meziprocesorové komunikaci uvnitř systému. Rozhraní se postupně vyvynulo až do součásti POSIX specifikace [33]. Vzhledem k celosvětovému rozšíření je v podstatě standardem pro síťovou komunikaci a obyčejně bývá přímo součástí operačních systémů.

Popis BSD socketů je abstrakce nad různými druhy spojení. V části POSIX standardu o socketech[7] můžeme nalézt poměrně nemálo věcí, které jsou specifikovány. Stěžejní z pohledu nástroje DIVINE a výběru vhodného komunikačního rozhraní jsou především dvě pasáže, a to `Address Famillies` a `Socket Types`. První definuje, skrz které médium se budou sockety používat, zatímco druhá podstatná pasáž je o tom, jaké vlastnosti bude mít samotný přenos dat skrz sockety.

Ačkoliv se v odkazované části POSIX standardu hovoří o "Address Families", dále ve standardu v části popisující funkce a jejich parametry se již hovoří o komunikační doméně. Budu tento termín nadále používat, neboť dle mého soudu lépe popisuje danou skutečnost.

2.2.1 Komunikační doména

POSIX standard popisuje konkrétně tři možné komunikační domény – lokální unixové sockety, sockety určené pro síťový protokol IPv4 [2] a sockety pro síťový protokol IPv6 [51]. Standard se pak dále zabývá technickými detaily, jako které číselné konstanty jsou určeny pro kterou rodinu adres, či jak přesně jsou zápisy adres reprezentovány v paměti.

Pro adresaci unixových socketů se používá cesta adresářovým stromem[8], s tím, že standard nedefinuje maximální délku cesty. Žádné další aspekty adresování nejsou při práci unixovými sockety zohledňovány.

V případě socketů pro síťový protokol je třeba rozlišovat mezi protokolem IPv4 a IPv6. První zmiňovaný používá k adresaci čtyři čísla v rozsahu 0 – 255. Při textové reprezentaci se ony čtyři čísla zapíšou v desítkovém základu a oddělí se tečkami. Ve strojovém zápisu se pak jedná o 32bitové číslo. Protokol IPv6 [24] reflektuje primárně nedostatek adres IPv4[9], což bylo při návrhu

[7] http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10_06

[8] Zde je nejspíše nutné malou poznámkou zmínit, že v souborových systémech unixového typu není adresářový strom stromem z pohledu teoretické informatiky, neboť umožňuje vytvářet cykly.

[9] Zpráva organizace ICANN: https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf

zohledněno a pro adresaci se používá osm čísel v rozsahu 0 – 65 535. Preferovaná textová reprezentace je zapsat oněch osm čísel v šestnáctkovém základu a oddělit je dvojtečkami. Ve strojovém zápisu se pak jedná o 128bitové číslo.

Vzhledem k tomu, že v současné době oba protokoly koexistují vedle sebe a nelze bez podrobnějšího zkoumání rozhodnout, zda nějaká síť používá IPv4, či IPv6, je nutné v rozhraní specifikace POSIX nabídnout takovou funkcionalitu, která bude (pro běžné používání) nezávislá na verzi protokolu IP. Touto funkcí je `getaddrinfo`, která přeloží jméno cílového stroje a/nebo jméno služby a vrací seznam adres, lhostejno zda IPv4, či IPv6, které lze dále použít ve funkcích řešících otevření spojení, zaslání dat a jiných.

Jméno cílového stroje obyčejně bývá název stroje v síti, což může zahrnovat i nadřazené domény. Více informací lze hledat v [42, 43][10] a [54][11]. Co je jméno služby bude probráno více u typů socketů.

2.2.2 Typy socketů

POSIX standard definuje čtyři různé typy: `SOCK_DGRAM`, `SOCK_SEQPACKET`, `SOCK_RAW` a `SOCK_STREAM` s tím, že implementace může definovat další typy. Ne všechny typy socketů jsou k dispozici ve všech komunikačních doménách, navíc je nutné mít na paměti, že výsledné chování každého typu může navíc záležet na komuniační doméně. Pro nástroj DIVINE je navíc klíčové použití síťové komunikační domény, takže pozornost bude věnována primárně na chvání různých typů socketů v této doméně.

Stručné vysvětlení, co jednotlivé typy socketů znamenají:

- `SOCK_STREAM` označuje spojité sockety.
- `SOCK_SEQPACKET` označuje sekvenční sockety.
- `SOCK_DGRAM` označuje nespojité sockety.
- `SOCK_RAW` označuje socket, který je použit pro implementaci výše uvedených typů socketů. Nazvěme je jako hrubé sockety.

Spojité sockety

Jak vyplývá z názvu, spojité sockety vytváří spojení mezi dvěma komunikujícími stranami, přičemž spojení musí být ustanoveno před započetím výměny dat. POSIX standard dále vyžaduje, aby v případě zaslání dat bylo garantováno doručení nepoškozených dat ve správném pořadí. Další důležitá vlastnost je, že nelze klást omezení na velikost poslaných dat. Je tedy možné si spojité sockety představovat jako dvojtou linku, pomocí které lze odesílat libovolná data a ze které lze zároveň libovolná data přijímat.

[10] Definují koncept doménových jmen a popisují jejich implementaci.
[11] Rozšíření doménových jmen pro IPv6.

Síťová doména V síťové doméně je zbytečné rozlišovat mezi IPv4 a IPv6. Novější IPv6 sice přidává některé zajímavé vlastnosti jako například možnost šifrování, ale obecně lze z pohledu spojitých socketů brát IPv4 a IPv6 za totožné. Spojité sockety jsou v síťové komunikační doméně implementovány pomocí TCP [3].

TCP implementuje všechny požadavky na spojité sockety, které klade POSIX standard. Před použitím je třeba navázat spojení, které operační systém provede ve třech krocích. V TCP je proud dat simulován tak, že jednotlivé bloky dat (dále jako záznamy) jsou vloženy do paketů. V případě, že je záznam příliš velký na paket, je záznam rozdělen mezi více paketů.

Garance neporušení dat je realizována kontrolním součtem, který je předáván společně s daty v hlavičce paketu. Garance doručení a doručení ve správném pořadí je v TCP prováděna tak, že každý odeslaný paket má své unikátní číslo, případně pořadové číslo (v případě rozdělení záznamu do více paketů), a odesílající strana vyžaduje potvrzení přijetí od přijímající strany do určitého času [55]. Pokud by k potvrzení nedošlo, je paket odeslán znovu. Na straně příjemce je v případě rozdělení záznamu do více paketů záznam zrekonstruován.

Zde je nutné zmínit se o portech. Pro úspěšné navázání komunikace je zapotřebí dvou různých entit – serveru (stroje, který očekává příchozí spojení) a klienta (stroje, který iniciuje komunikaci). V rámci serveru je obyčejně více aplikací, které čekají na příchozí spojení. Každé takové čekání musí být jednoznačně identifikováno a k tomu slouží port – číslo, které popisuje jeden konec (ještě nenavázaného) spojení.

Klient musí dopředu vědět, na kterém portu chce na serveru navázat spojení, proto byl v počátcích internetu zveřejněn dokument [49], který byl postupně aktualizován[12] a obsahoval původně seznam 256, posléze až 1024, obsazených portů pro specifikované účely, například porty 20 a 21 pro FTP [4] nebo port 80 pro web a HTTP [26]. Číslo portu je v síťové doméně jméno služby a je předáváno jako parametr do funkce `getaddrinfo`.

Unixová doména V unixové doméně lze spojité sockety chápat jako speciální soubory, které mohou být podobně jako soubory `FIFO`[13] použity pro meziprocesovou komunikaci. Spojité sockety mají navíc od `FIFO` ty vlastnosti, že jsou obousměrné a že je možné je použít pro přenos zdrojů mezi procesy (například otevřené popisovače souborů).

Realizace garancí spojitých socketů je v případě unixové komunikační domény jednoduchá, za předpokladu korektní implementace a korektních prvků hardware, jako například pamětí, či pevných disků.

[12] V roce 2002 bylo původní [49] nahrazeno za [48], které odkazuje na online databázi na adrese `http://www.iana.org/assignments/protocol-numbers/`.

[13] `FIFO`, neboli také `pipe` – trubky, roury – jsou speciální soubory, které se chovají jako jednosměrný proud dat. Jsou typicky na jednom konci otevřeny pro zápis a na druhém konci pro čtení.

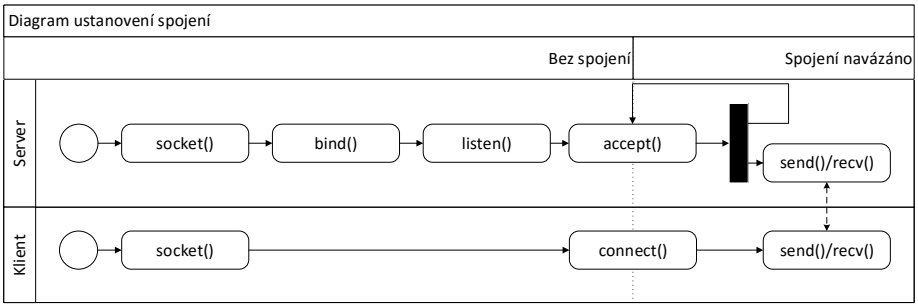
Ukázka použití Z pohledu použití spojitých socketů v programu šíťové komunikační doméně je potřeba odlišit kroky, které je třeba vykonat na straně serveru a na straně klienta pro navázání spojení.

Server

1. Vytvoření socketu pomocí funkce `socket`.
2. Zamluvení si portu pomocí funkce `bind`.
3. Nastavení socketu jako připraveného a nastavení maximální fronty čekajících spojení funkcí `listen`.
4. Čekání na příchozí spojení na připraveném socketu pomocí funkce `accept`.
5. Po přijetí a zpracování příchozího spojení opakovat krok 4.

Klient

1. Vytvoření socketu pomocí funkce `socket`.
2. Požádání o spojení funkcí `connect`.



Obrázek 2.1: Diagram ustanovení spojení

Po navázání spojení si jsou klient i server rovni a oba mohou používat funkce pro odesílání i přijímání dat. Spojité sockety je možné na libovolné straně kdykoliv uzavřít, či přivřít. Zavřít socket je možné voláním funkce `shutdown`, která v případě TCP provede čtyřkrokové rozvázání spojení. Standardním voláním funkce `close` dojde k uvolnění popisovače souboru, ale až poslední volání nad daným socketem funkce `close` zavolá funkci `shutdown`.

Sekvenční sockety

Sekvenční sockety se od spojitých socketů mnoho neliší. Vytváří spojení mezi dvěma komunikujícími stranami a spojení musí být stejně tak ustanoveno před započetím výměny dat. Zůstávají stejné garance, jaké POSIX standard vyžaduje od spojitých socketů – garance doručení a garance doručení nepoškozených dat.

V čem se ale liší, je sémantika přenosu dat. Spojité sockety vytvářely iluzi proudu dat, který hypoteticky nemusí skončit. Naproti tomu sekvenční sockety mají sémantiku jednotlivých zpráv. Každý blok dat, který je poslaný sekvenčním socketem, má nějakou omezenou velikost a příjemce na druhé straně si musí nejprve vyzvednout celý záznam, než může přijmout další zprávu. Funkce pro příjem zprávy tedy dokáží rozeznat hranice jednotlivých zpráv.

Síťová doména POSIX standard nespecifikuje, jak má být tento typ socketu implementován v síťové komunikační doméně. Ačkoliv je možné předpokládat, že bude tento typ socketu dostupný v proprietárních sítích, v běžně používaných síťových prvcích není dostupný.

Unixová doména Sekvenční sockety lze v unixové komunikační doméně taktéž chápat jako speciální soubory obdobně jako spojité sockety, přičemž z pohledu souborového systému jsou dokonce totožné. Implementaci sekvenčních socketů si není možné představovat jako obousměrné `FIFO` soubory, ale spíše jako frontu samostatných zpráv či balíčků dat. Obdobně jako u spojitých socketů je garance spolehlivosti závislá na korektní implementaci a bezchybnosti hardware.

Je vhodné na tomto místě podotknout, že ačkoliv jsou sockety implementovány pomocí souborů, nemusí to nutně znamenat, že klient i server musí běžet na stejném stroji. Souborový systém může být například sdílený v rámci vnitřní sítě.

Ukázka použití Použití je vesměs totožné jako u spojitých socketů, ovšem s tím rozdílem, že namísto portu je potřeba uvést cestu v adresářovém stromě, čímž se ve specifikované složce vytvoří speciální soubor typu sekvenční socket. Aplikace, které chtějí znovupoužít ten samý socket, pak musí řešit problém kdy smazat soubor – jestli před zamluvením socketu, nebo při skončení programu.

Vzhledem k nemožnosti použít sekvenční sockety v síťové komunikační doméně je ukázka platná pro unixovou komunikační doménu. Zde je uvedena varianta z pohledu serveru, kdy se případný soubor smaže při startu.

Server

1. Vytvoření socketu pomocí funkce `socket`.
2. Smazání případného souboru funkcí `unlink`.
3. Zamluvení si portu pomocí funkce `bind`.
4. Nastavení socketu jako připraveného a nastavení maximální fronty čekajících spojení funkcí `listen`.
5. Čekání na příchozí spojení na připraveném socketu pomocí funkce `accept`.
6. Po přijetí a zpracování příchozího spojení opakovat krok 5.

Nespojité sockety

Dalším typem socketů jsou nespojité sockety. Na rozdíl od dříve popisovaných typů socketů neslouží k vytváření spojení a zároveň nedávají žádné garance na přenášená data. Protože POSIX standard vyžaduje, aby ke každé operaci zápisu do nespojitého socketu náležela právě jedna operace čtení ze socketu, může dojít dokonce k tomu, že ačkoliv jsou data úspěšně doručena, může příjemce chybně nastavenou operací čtení ze socketu způsobit zahození části dat.

Garance na nespojité sockety jsou dokonce tak slabé, že dle POSIX standardu je legitimní data vůbec neodeslat a zároveň nereportovat žádnou chybu – například protože operační systém zapsal data do vyrovnávací paměti, vrátil řízení zpět programu a až poté se neúspěšně pokusil data odeslat. Toto chování by ovšem nemělo být časté.

Protože nespojité sockety nevytváří spojení, celý komunikační model je diametrálně odlišný od předchozích dvou typů socketů. Každý potenciálně komunikující program si musí vytvořit socket, na kterém bude poslouchat, a který se ze sémantického hlediska chová spíše jako poštovní schránka. Co se týká odesílatele, jedinou možností, jak zjistit odkud byla zpráva poslána, je uložit si adresu při přijetí zprávy. Tato možnost je sice možná také u spojitých a sekvenčních socketů, nicméně je poněkud zbytečné ji používat, neboť socket nemůže během své existence změnit ani jednu komunikující stranu.

Síťová doména Podobně jako u spojitých socketů je v kontextu síťové domény zbytečné rozlišovat mezi IPv4 a IPv6. Jediný podstatnější rozdíl bude zmíněn později. Nespojité sockety jsou v síťové komunikační doméně implementovány pomocí UDP [46].

UDP slouží k co nejjednoduššímu předávání dat od odesílatele k příjemci. Z tohoto důvodu mechanizmy, jako například garance doručení, neporušení integrity dat a jiné, které jsou implementované v TCP, nejsou v případě UDP dostupné. Shodně s TCP je v případě UDP přenos realizován pomocí paketů.

Vzhledem k minumu protokolových mechanizmů je hlavička UDP paketu velmi krátká a obsahuje pouhé čtyři údaje:

1. Zdrojový port
2. Cílový port
3. Délka dat
4. Kontrolní součet dat.

Zde je zajímavá položka kontrolního součtu dat. Dle [46] sice může hlavička paketu obsahovat kontrolní součet dat, ovšem nikde není specifikováno, co se má dít v okamžiku, kdy příjemce zjistí, že se kontrolní součet přijatých dat neshoduje se součtem uvedeným v hlavičce.

Další zajímavou položkou z hlavičky UDP paketu je délka dat. Při znalosti této informace by bylo možné připravit přesně velkou paměť, do které by se

data přijala. Rozhraní, které popisuje POSIX standard[14] ale takovou možnost nenabízí, resp. funkce vrací tuto hodnotu až poté, co došlo k přečtení zprávy. Je tedy nutné při navrhování protokolu, který bude používat UDP, zvážit nejvyšší možnou délku zprávy.

Vzhledem k tomu, že UDP nevytváří spojení, je nutné při každém zaslání zprávy specifikovat příjemce. Jedním z vedlejších důsledků této vlastnosti je, že je možné zprávu zaslat na speciální adresu, což způsobí, že se zpráva rozešle všem síťovým prvkům v lokální síti. V IPv4 se tento způsob doručování zpráv jmenuje *broadcast* a ona speciální adresa je `255.255.255.255`. V IPv6 se nehovoří o broadcastu, ale o *multicastu*[15]. Multicast se od broadcastu liší tím, že zpráva nemusí být zaslána všem strojům v místní síti, ale pouze těm, kteří se nahlásí o její přijetí. Zpráva zaslaná pomocí multicastu navíc může být doručena i mimo lokální síť.

Unixová doména Stejně jako předchozí typy socketů jsou i nespojité sockety reprezentovány pomocí souboru v souborovém systému. Ačkoliv i v unixové komunikační doméně platí stejně slabé garance, lze si například pohledem do zdrojového kódu implementace GNU/Linuxu[16] verze 4.3 všimnout, že například zápis do sekvenčního socketu je implementován pomocí stejné funkce jako zápis do nespojitého socketu. Z toho vyplývá, že nespojité sockety někdy mohou mít stejné garance jako sekvenční sockety.

Ukázka použití Oproti předchozím dvěma ukázkám použití je tento typ socketů diametrálně odlišný. Vzhledem k faktu, že nedochází k ustanovení spojení, zde není rozdělení na server a na klienta, ale pouze na komunikující subjekty.

Ukázka použití v unixové komunikační doméně[17]:

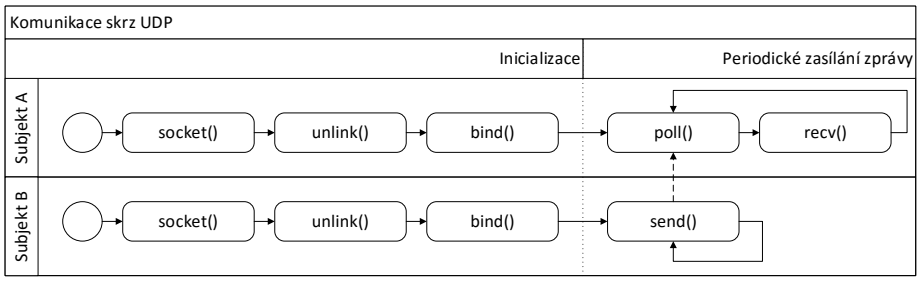
1. Vytvoření socketu pomocí funkce `socket`.
2. Smazání případného souboru funkcí `unlink`.
3. Zamluvení si portu pomocí funkce `bind`.
4. Čekat na příchozí zprávu pomocí funkcí `select` či `poll`.
5. Přečíst a zpracovat zprávu.

[14] Jedná se primárně o funkce `recv`, `recvfrom` a `recvmsg`.

[15] I v IPv4 byl později [25, 14] také implementován multicast, ovšem koncovému uživateli Internetu nemusí být dostupný.

[16] Funkce `unix_dgram_sendmsg` a `unix_seqpacket_sendmsg` na `http://lxr.free-electrons.com/source/net/unix/af_unix.c`.

[17] Pro nasazení do síťové komunikační domény by postačilo vynechat krok 2 – smazání případného souboru.



Obrázek 2.2: Komunikace skrz UDP

Hrubé sockety

Hrubé sockety[18] jsou posledním typem socketů, které definuje POSIX standard. Sami o sobě nejsou samostatným typem (pokud by byly srovnávány s třemi předchozími), jsou spíše základem pro dříve jmenované. Žádostí o vytvoření hrubého socketu program žádá rozhraní operačního systému pro přímý přístup ke třetí vrstvě ISO/OSI [58] modelu.

Touto cestou je možné implementovat libovolný vlastní protokol na čtvrté vrstvě ISO/OSI modelu. Moderní operační systémy z důvodu bezpečnosti nedovolují aplikacím s běžným oprávněním vytváření takovýchto typů socketů, proto se jimi nebudu v této práci dále zaobírat.

2.3 Asio

Asio [36] je knihovna z Boostu, která nabízí asynchronní vstupně-výstupní operace. Je také jednou z knihoven, které mohou být použity i samostatně[19].

2.3.1 Boost

Boost[20] je seskupení C++ knihoven, které pokrývají mnoho témat, od základních věcí jako časovače, přes statistické funkce, až po práci s obrázky či regulárními výrazy. Celkově tak Boost obsahuje několik desítek takových knihoven. Některé knihovny navíc mohou existovat ve dvou formách – jednou jako součást Boostu, podruhé jako samostatná knihovna.

Mnohé knihovny z Boostu, obdobně jako nemalé části Standardní C++ knihovny[21] (dále jako STD), nabízejí sjednocené rozhraní k systémovému

[18] V originálu *raw sockets*.

[19] http://think-async.com

[20] http://www.boost.org

[21] Standardní C++ knihovna je kolekce tříd a funkcí, které jsou napsány převážně v jazyce C++ (některé části, například Knihovna jazyka C, jsou psány v jazyce C). Standardní C++ knihovna je součástí ISO C++ standardu, který tak definuje rozhraní a další vlastnosti knihovních tříd a funkcí. Více o standardní knihovně lze nalézt na http://en.cppreference.com/w/cpp/links a v odkazech uvedených na stránce.

rozhraní, které se může lišit na základě operačního systému. Jako příklad co nejrozdílnějšího systémového rozhraní lze uvést operační systémy Microsoft Windows a UNIX-like[22] systémy. Knihovny z Boostu, na rozdíl od STD, mnohdy poskytují specifická rozšíření pro některé architektury a operační systémy.

Vztah knihoven obsažených v Boostu a STD je v některých případech velmi těsný. Důvod je ten, že spousta tříd a funkcí, které jsou standardizovány a tedy jsou součástí STD, mají svůj původ v některé knihovně v Boostu. V novějších verzích standardu jazyka C++ pak těchto knihoven přibývá.

2.3.2 Knihovna Asio

Hlavním přínosem knihovny Asio je její pojetí asynchronních volání. Asynchronní volání se v posledních několika letech stávají populární především možností použít ho v rozšířených programovacích jazycích, jako je například Java[23] nebo C#[24]. Také v poslední velké revizi jazyka C++ se asynchronní volání objevily – v podobě funkce `std::async`. Asynchronní volání ve zkratce znamená, že namísto běžného volání funkce poznačíme, že požadujeme provedené té které funkce, a až v místě, kde potřebujeme znát výsledek, si o něho požádáme. Je pak na možnostech jazyka a běhového prostředí, aby se postaralo o vyhodnocení asynchronně volané funkce. Možností je zde více, namátkou například spuštění asynchronní funkce v samostatném vlákně, nebo prolnutí funkcí během překladu do strojového kódu či mezikódu.

Další neméně důležitou součástí jsou vstupně-výstupní operace. STD poskytuje rozhraní pro práci se standardním vstupem a výstupem a také rozhraní pro práci se soubory. Co už ale STD knihovna nenabízí, je práce se sítí, kterou naopak knihovna Asio poskytuje. Poskytnuté funkce a třídy jsou navíc psány v obdobných konvencích jako STD.

Rozhraní pro síťovou komunikaci implementuje knihovna Asio nad BSD sockety. Samozřejmě poskytuje podporu pro použití obou verzí IP, taktéž pro TCP i UDP. Navíc, protože je v současné době často nutnost použít zabezpečené spojení, umožňuje knihovna Asio použít SSL [8], ke kterému je ale potřeba další knihovna – OpenSSL[25].

2.3.3 Princip používání Asio

Knihovna Asio je postavena na návrhovém vzoru Proaktor [52]. Základem pro veškeré asynchronní operace je objekt třídy `asio::io_service`, který

[22] Jedná se o operační systémy, které vycházejí z filozofie systému UNIX [50], poskytují obdobné rozhraní, ale neprošly procesem standardizace. Jako příklad lze uvést operační systémy jako BSD, či System V.

[23] https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/FutureTask.html

[24] https://msdn.microsoft.com/en-us/library/hh873175%28v=vs.110%29.aspx

[25] https://www.openssl.org/

zároveň plní úlohu proaktora. Další třídy, které reprezentují například sockety, mají roli inicializátorů. Poslední rolí, která stojí za zmínku, je oznamovač, což je funkce, která je spuštěna proaktorem po dokončení asynronní operace.

Asynchronní režim se za podpory knihovny Asio používá poměrně jednoduše. Po vytvoření instance třídy `asio::io_service` program zaregistruje různé události, které ho zajímají. Může to být impulz z časovače, příchozí spojení nebo třeba dokončený zápis dat do souboru. Současně s registrací program upřesňuje, jaké funkce se mají zavolat v okamžiku, kdy událost nastane. Jediné, na co je třeba dávat pozor, je opětovná registrace na nastalou událost, protože spuštěním obsluhy události dojde k odregistrování. Po zaregistrování všech požadovaných událostí stačí spustit metodu `run` na proaktoru.

Třídy z knihovny je možné použít i pro synchronní vstupně-výstupní operace, pokud budou jejich metody volat přímo a nikoliv skrze proaktora. Protože ale spousta tříd ke svému vytvoření vyžaduje objekt třídy `asio::io_service`, jejímu vytvoření se není možné nevyhnout.

2.4 Stávající komunikační rozhraní

V nástroji DIVINE je od konkrétního rozhraní abstrahováno pomocí rozhraní, která se stará o topologii. Topologie zajišťuje správné nastavení výkonných jednotek, ať už vláken, nebo procesů, stará se o realizaci komunikace a vzájemnou synchronizaci výkonných jednotek. Implementovány jsou konkrétně dvě. Jedna reprezentuje lokální stroj a její starostí jsou vlákna. Druhá reprezentuje skupinu procesů, takže její starostí jsou procesy, uvnitř nichž poté deleguje zodpovědnost za vlákna lokální topologii.

Topologie reprezentující skupinu strojů je napojena na rozhraní MPI funkcí[26]. Z nich jich používá jen několik. Používá převážně neblokující odesílání zpráv, taktéž zjišťování příchozích zpráv provádí neblokujícně, samotný příjem zprávy zůstává blokující.

Ačkoliv standard MPI říká o možnosti paralelního přístupu ke svému rozhraní, ne všechny implementace MPI tuto možnost nabízejí a konkrétně implementace, kterou používáme – Open MPI – v dokumentaci uvádí, že paralelnímu přístupu nevěnují dostatek pozornosti a spíše se soustředí na korektnost než na výkon. Nástroj DIVINE proto používá dedikované vlákno, které obsluhuje veškerou MPI komunikaci. Ostatní vlákna, pokud chtějí odeslat data jinému procesu, je musí nejprve vložit do fronty na zpracování vláknu dedikovanému na komunikaci. Obdobně je tomu tak i v případě příjmu zprávy, kdy dedikované vlákno vkládá příchozí zprávy výkonným vláknům do jejich front na zpracování. Dohromady tedy dochází ke zbytečnému kopírování bloků dat, kterému se ovšem nelze při použití MPI efektivně vyhnout.

[26] V tomto případě by bylo lepší hovořit o metodách, protože nástroj DIVINE používá rozhraní C++ ze standardu MPI-2.

V nástroji DIVINE je komunikační rozhraní využito pouze při spuštění v hybridním režimu paralelizace. Sama hybridní paralelizace si s sebou nese jisté potíže, které jsem v první kapitole přednesl. Jeden z problémů, kterým je zapojení komprese paměti bez zvýšené režie, by bylo možné vyřešit, pokud by se řízení vláken při procházení grafu realizovalo obdobně, jak se realizuje v režimu paralelizace ve sdílené paměti. To by navíc mohlo vést i k snížení nerovnoměrného rozložení zátěže. Tento koncept, pracovně nazvaný jako dvouvrstvá architektura, jsem předestřel již dříve v své bakalářské práci, ovšem z důvodu upřednostnění jiných úkolů nebyl doposud realizována.

Kapitola 3

Návrh a implementace nového komunikačního rozhraní

Při zvažování, zda implementovat vlastní zjednodušenou nadstavbu nad BSD sockety, nebo zda použít knihovnu Asio, jsem zvolil první možnost. Jako důvod uvádím, že použití knihovny Asio by zavedlo do projektu další závislost – buď ve formě správné verze knihovny Boost na straně uživatele, nebo ve formě nutnosti dodávat zdrojové soubory knihovny Asio spolu se zdrojovými soubory nástroje DIVINE, přičemž obojí přináší režii do správy projektu, který je limitován lidskými zdroji. Druhým důvodem pak může být, že z možností knihovny Asio by byla v nástroji DIVINE využita jen malá část.

Pro samotnou implementaci pak bude potřeba ze dříve uvedených typů socketů vybrat ten nejvhodnější. Nová implementace dále vyžaduje vytvořit jednoduchý komunikační protokol pro ustanovení sítě strojů kooperujících na distribuovaném výpočtu. Při návrhu nové implementace se navíc nemusím držet architektury distribuované aplikace, jak ji popisuje MPI, která má dle mého názoru některé vady, pročež jsem se rozhodl, že vytvořím architekturu s jinými vlastnostmi.

Nová implementace bude nasazena na výpočetním stroji, jehož architektura je x86-64 [1] a jehož operační systém používá rozhraní definované POSIX standardem – převážně počítám s operačním systémem GNU/Linux[1]. Dále předpokládám, že endianita všech strojů participující na výpočtu je stejná, což je předpoklad, který využiji při návrhu formy zasílání dat.

3.1 Analýza vlastností typů socketů

Pro volbu správného typu socketů je třeba pečlivě zvážit vlastnosti jednotlivých typů, neboť následná implementace se může velmi lišit. Cílem je implementovat síťovou komunikační vrstvu do nástroje DIVINE, takže nebudu příliš rozebírat vlastnosti socketů v unixové komunikační doméně.

[1] http://www.gnu.org/gnu/linux-and-gnu.html.en

3.1.1 Spojité sockety

Spojité sockety mají několik vlastností, ať už dobrých, či horších, které je třeba vzít v potaz:

- udržování spojení
- možnost mít více spojení mezi výpočetními stroji
- garance doručení nepoškozených dat
- možnost poslat neomezeně dlouho zprávu
- režie spojená s udržováním spojení a garancemi

Udržování spojení

Distribuované výpočty, které nástroj DIVINE provádí, vyžadují, aby žádný z participatů výpočtu nepřerušil kontakt s ostatními. To se může stát jednak chybou v síti – rozpojením sítě – jednak zastavením výpočtu na stroji, ke kterému může dojít například z důvodu chyby v programu. U spojitých socketů jsou při přerušení spojení notifikování oba participanti komunikace, tudíž může dojít k následnému korektnímu ukončení distribuovaného výpočtu.

Nevýhodou je nutnost pravidelné výměny zprávy, která říká, že spojení je stále aktivní, mezi participanty. Tato výměna je ale nezbytná pouze v případě, že v mezičase nebyla mezi participanty komunikace zaslána žádná zpráva. Vzhledem k předpokladu, že mezi výpočetními stroji bude docházet k čilé výměně dat, nepokládám tuto nevýhodu za kritickou.

Více spojení

Mezi dvěma komunikujícími stroji je v případě spojitých socketů možné navázat více než jedno spojení v rámci jednoho obslužného portu na straně serveru. Tento aspekt je možné vypozorovat ze sekvence kroků, které vykonává server k přijímání a zpracování příchozách zpráv, které byly prezentovány u příkladu použití. Ustanovení více souběžných spojení může být potenciálně výhodné, neboť každé takové spojení můžeme chápat jako komunikační kanál a každý kanál použít pro jiný účel.

Lze tak například vytvořit jeden kanál pro posílání řídících zpráv, další kanály pak pro posílání dat. Výhodou navíc může být, že každé vlákno nástroje DIVINE bude obsluhovat svůj vlastní datový kanál bez nutnosti explicitního zamykání přístupu ke komunikačnímu zdroji.

Vzhledem k tomu, jak nástroj DIVINE pracuje v distribuovaném výpočtu s daty, které posílá na zpracování a které naopak přijímá, a vzhledem k očekávaným velikostem datových balíčků, je výhodné se vyhnout častému kopírování dat. Přístup více spojení přenechává operačnímu systému rozhodování o přidelených vyrovnávacích pamětech, což se může projevit v efektivnější práci s pamětí, protože bude docházet k menšímu počtu nutného kopírování bloků dat.

Garance doručení nepoškozených dat

Nástroj DIVINE potřebuje ke své práci korektní data. V případě běhu ve sdílené paměti je jednoduché zaručit, že se při manipulaci s daty nestane, aby byly nějak poškozeny. V distribuovaném prostředí, kdy je pro přenos dat mezi výpočetními uzly třeba použít síťové rozhraní, nastupuje mnoho hardwarových prvků, z nichž každý může ovlivnit konzistenci přenášených dat.

Spojité sockety, konkrétně implementované pomocí TCP, tyto garance zajišťují pomocí kontrolních součtů dat při příjmu. Pokud je kontrolní součet dat odlišný od toho, který byl v hlavičce TCP paketu, dojde k opětovnému přenosu paketu. Použitím spojitých socketů tak odpadá potřeba implementovat vlastní mechanizmy do nástroje DIVINE, které by zaručovaly nepoškozená data.

TCP sockety mají navíc mechanizmus, který zaručuje (v případě, že to je možné) doručení dat. Znamená to tedy, že pokud do určitého časového intervalu nedorazí odesílateli potvrzení o doručení, pošle odesílatel zprávu znovu.

Možnost poslat neomezeně dlouhou zprávu

TCP vytváří abstrakci, že přenášená data jsou souvislý proud dat. Protože TCP pakety mají omezenou maximální velikost, dochází při poslání velkého bloku dat k rozdělení záznamu do více paketů, které jsou samostatně poslány sítí a na straně přijemce opět poskládány. Na toto se také vztahuje garance nepoškozených dat, která je implementována tak, že jsou pakety tvořící jeden blok dat sekvenčně očíslovány, takže případně prohození pořadí paketů zachytí příjemce a dle čísel poskládá správné pořadí.

Tato vlastnost TCP je z jedné strany výhodou, protože není třeba v samotné aplikaci řešit, zda se data vejdou do paketu, či zda je nutné je rozdělit a posléze spojit – výhodou je, že není třeba implementovat vlastní mechanizmus. Nevýhodou pak může být případně zdržení, neboť ztracený či poškozený paket zdržuje nejen celou zprávu, ale celý jeden směr komunikačního spojení, protože nemůže být přenesen jiný, bezproblémový blok dat.

Režie spojená s garancemi

Předem zmíněné výhody vyvažuje na druhé straně režije spojená jak s udržováním spojení, tak se zasíláním opravných paketů. Udržování spojení je realizováno periodicky se opakujícím posíláním krátkého TCP paketu mezi komunikujícími stranami. Toto vyměňování zpráv má nějakou nezanedbatelnou režii, ovšem posílání krátkých paketů je nutné pouze v případě, že ono spojení nebude řádně vytěžováno – každý poslaný TCP paket je totiž známkou toho, že spojení je stále udržováno.

Co se týká garancí na doručení, vyvstávají zde dva potenciální problémy. První z nich je, že operační systém si musí držet v paměti již odeslaná

data, a to z důvodu případného opakování přenosu. Může tedy docházet ke zbytečnému využívání paměti. Druhým problémem je nutnost potvrzení doručení paketu. Toto je opět možné realizovat samostatným krátkým paketem v případě zřídkavého datového provozu v opačném směru, nebo přidáním příznaku doručení do hlavičky paketu jdoucím po stejné lince zpět.

Z krátké analýzy vyplývá, že v případě zřídkavého zasílání dat dochází k režii, která zatěžuje nejen oba komunikující subjekty, ale také veškerou síťovou infrastrukturu mezi nimi. U nástroje DIVINE se nicméně očekává, že datový provoz mezi všemi participanty výpočtu bude velmi vysoký, takže síť nebude zbytečně zaplavována krátkými pakety. Každopádně platí, že v porovnání s UDP paketem má TCP paket výrazně větší hlavičku a tedy množství dat přenesených po čas komunikace pomocí TCP mezi dvěma subjekty úměrně tomu vzrůstá oproti komunikaci přes UDP.

3.1.2 Nespojité sockety

Nespojité sockety také nabízejí zajímavé vlastnosti, které je vhodné rozebrat:

- žádné garance
- jednoduchý protokol
- omezení na velikost zprávy
- neudržování spojení

Ačkoliv mají nespojité sockety pokud možno vyvinout co největší úsilí k tomu, aby zpráva došla jejímu příjemci, nejsou dané žádné garance o tom, jestli data dojdou v pořádku, či zdali vůbec dojdou. Toto s sebou nese dva podstatné důsledky.

Jednak to znamená, že se bude v síťové komunikaci posílat méně dat v hlavičkách, což znamená, že by za stejný časový interval bylo možné poslat více zpráv. To navíc souvisí s další vlastností nespojitých socketů, kterou je jednoduchý protokol.

Pak to také znamená, že by bylo potřeba implementovat vlastní protokol nad UDP pro ověřování korektnosti dat a pro ověřování jejich doručení. Předpokládá se, že nástroj DIVINE bude provozován v uzavřených sítích, takže je zde silný předpoklad na to, že síťových prvků mezi dvěma komunikujícími stroji bude malý počet – typicky jeden síťový rozbočovač. Případně implementovaný protokol by tedy měl být šetrný k běžnému provozu a až v případě chyby při přenosu by mělo dojít k vyšší režii.

Další vlastností nespojitých socketů je maximální délka zprávy. Maximální bezpečnou délku UDP paketů omezuje [12] na 576 bytů minus délka IP hlavičky, což v praxi omezuje délku UDP paketu na 512 bytů. UDP paket může být větší, pokud všechny síťové prvky na cestě mezi odesílatelem a příjemcem podporují delší zprávy. Protože nástroj DIVINE posílá zprávy o velikosti několika málo desítek kilobytů, znamenalo by to potřebu zajistit fragmentaci dat na straně odesílatele a defragmentaci na straně příjemce.

Poslední vlastností, kterou jsem výše uvedl, je neudržování spojení. Zde je namístě krátká polemika o tom, jak je síťová komunikace využívaná z pohledu nástroje DIVINE. Ačkoliv od komunikační vrstvy DIVINE požaduje v podstatě pouze posílání zpráv, kde by se nespojité sockety hodily, potřebuje také udržovat znalost o tom, že jsou všechny stroje participující na distribuovaném výpočtu spojené. Danou funkcionalitu UDP nenabízí a bylo by tedy nutné ji taktéž implementovat.

Požadavky pro protokol

Z výše uvedených vlastností UDP a nespojitých socketů vyplývají některé vlastnosti, které by případně implementovaný protokol měl zajišťovat. Jsou jimi:

1. garance doručení zpráv
2. garance doručení nepoškozených dat
3. rozdělení dlouhé právy a její složení při příjmu
4. detekce, zda jsou stroje stále spojeny

Složení dlouhých zpráv z více UDP paketů v sobě ukrývá příjemnou vlastnost, kterou je nutnost použití další vrstvy vyrovnávacích pamětí a s tím spojenou režii ve formě kopírování bloků dat z a do těchto pamětí. Tato režie se navíc vyskytuje i na straně odesílatele – zde ovšem z důvodu garantování zaslání nepoškozených dat. Každý odeslaný blok dat musí být na straně odesílatele uchován tak dlouho, dokud neobdrží potvrzení o přijetí.

Při zamyšlením nad zapojením komunikační vrstvy do nástroje DIVINE bychom narazili na další problém, kterým je paralelní přístup k přichozím zprávám. Na jednom výpočetním stroji lze mít na jednom portu otevřenou pouze jednu komunikačním linku, což znamená mít buď jedno dedikované vlákno na veškerou obsluhu spojení, nebo implementovat synchronizaci pomocí zámků pro paralelní přístup k socketu.

Spolu s nutností použít další vrstvu vyrovnávacích pamětí se pak dedikované vlákno jeví jako snadnější volba, neboť by bylo potřeba vhodně zvolit granularitu zámků pro paralelní přístup – od jednoho globálního zámku, který je nevhodný z důvodu malé rychlosti a ztráty výhod paralelizmu, až po paralelní zámek pro každý blok paměti a socket, kde hrozí problémy typu uváznutí či porušení dat.

3.1.3 Sekvenční sockety

Sekvenční sockety zde uvádím jen pro ilustraci. Jejich použití totiž nepřipadá v úvahu, neboť nejsou definovány pro síťovou komunikační vrstvu. Stále ale stojí za zmínku některé jejich vlastnosti.

Garance

Sekvenční sockety poskytují stejné garance jako spojité sockety. Vzhledem k rozborům uvedeným výše se jedná o klíčovou vlastnost,kterou nástroj DIVINE od komunikační vrstvy požaduje. Tudíž by byly z toho pohledu vhodnými kandidáty.

Explicitní stanovení hranic zpráv

Oproti spojitým socketům pevně ohraničují zprávy, takže se nemůže stát, že následkem chybného zpracování dojde k posunutí hranice konce zprávy. Nicméně tato vlastnost není klíčovou pro komunikační vrstvu – jednalo by se pouze o přidání dalšího kontrolního mechanizmu.

3.1.4 Zvolené řešení

Na výběr zůstaly spojité sockety implementované pomocí TCP a nespojité sockety, které jsou implementované pomocí UDP. Obě dvě implementace jsou jako standardní síťové protokolyy na strojích typicky implementovány na úrovni operačního systému. V případě TCP to přináší několik výhod:

1. Veškerá obsluha spojení, příchozích dat a režie je prováděna v rámci jedné úrovně bezpečnosti procesoru [34] a nedochází tak k vícenásobnému přepínání kontextu z uživatelského prostoru (*ring 3*) do jádra operačního systému (*ring 0*).
2. Obsluha síťového rozhraní musí používat zámky pro paralelní přístup. Výhodou je, že u rozšířeného operačního systému, jakým GNU/Linux je, se očekává, že implementace nebude obsahovat problémy typu uváznutí a bude optimalizovaná.
3. Implementace TCP pracuje šetrně s pamětí. Znamená to, že obsahy příchozích paketů skládá do vyrovnávacích pamětí a až v případě, že přijde z uživatelského prostoru požadavek na přijetí dat, se zpráva zkopíruje do programem označené paměti. Příchozí zpráva je tedy dvakrát kopírována.
4. TCP je v provozu již více jak 30 let. Lze tedy očekávat, že případné chyby v protokolu a v implementacích operačních systémů již byly nalezeny a opraveny.

Oproti tomu použití nespojitých socketů umožňuje využít malou režii a zároveň vyšší rychlost díky kratším hlavičkám UDP paketů. Nevýhodou je nutnost implementovat vlastní protokol pro udržování spojení, pro garanci korektnosti dat a pro další dříve zmíněné požadované vlastnosti. Výhody UDP se rázem stávají nevýhodami:

1. Obsluha spojení, příchozích dat a režie je prováděna v uživatelském prostoru. Každý dotaz na příchozí data nutně vyžaduje přepnutí kontextu z uživatelského do jádra opračního systému, což je drahá operace vzhledem k času [40].
2. V případě paralelního přístupu by bylo potřeba navíc použít zámky pro paralelní přístup v uživatelském prostoru. Další zámky mohou hypoteticky snížit rychlost zpracování dat. V případě použití dedikovaného vlákna pro zpracování příchozích spojení pak musíme řešit navíc mezivláknovou komunikaci, kde se pravděpodobně také použijí zámky. Povšimněte si, že k zamykání systémových zámků dochází i při tomto přístupu.
3. Další vrstva vyrovnávacích pamětí, kterou je potřeba použít, způsobí, že bloky dat nebudou před přijetím kopírovány dvakrát, ale třikrát.
4. U čerstvě vytvořeného protokolu hrozí vyšší riziko chyb – jak návrhové chyby, tak implementační.

Z tohoto shrnutí se mi jeví jako vhodné použít spojité sockety spolu s TCP. Druhá možnost by znamenala dle mého názoru vyvýjet již vynalezenou věc znovu s nejistým výsledkem, zda bude rychlejší (a zda bude korektní).

3.2 Architektura

Stávající řešení, kdy se MPI agent postupně přes *ssh* [53] připojí ke všem strojům, kteří se mají účastnit distribuovaného výpočtu, provede se výpočet a jednotlivé instance na výpočetních strojích se ukončí, vypovídá o pohledu, jakým se díváme na nástroj DIVINE. Jako na program, který jednou spustíme, on nám dá výsledek a tím je vše hotovo.

Na nástroj DIVINE se ale můžeme dívat i jinak. Jako na službu, která běží někde ve výpočetním clusteru a která je dostupná skrz nějaké rozhraní – skrz grafickou aplikaci, příkazový řádek či webovou stránku. Pro tento nový pohled je ale potřeba provést změny v architektuře. Jedná se zejména o rozdělení programu na dvě části. První a podstatnější je serverová část, která je zodpovědná za samotné provedení výpočtu. Druhou částí je klientská část, která je v aktuálním návrhu zodpovědná za zadání práce vybraným serverům.

3.2.1 Server

Server bude mít jeden hlavní proces, jehož primárním úkolem je příjem příchozích spojení a odpovídání na požadavky. V případě započetí výpočtu dojde k vytvoření dalšího procesu, který je zodpovědný za vykonání výpočtu – tzv. výkonný proces. Tím hlavní proces změní svoji roli a stává se opatrovníkem výkonného procesu. V jeden okamžik může běžet nanejvýš jeden výkonný proces.

Primárním důvodem tohoto rozdělení je mít kontrolu nad kteroukoliv běžící instancí programu na vzdálených strojích. Tato funkcionalita je obzvláště vhodná v případě, že se distribuovaný výpočet zacyklí či dojde k uváznutí. Skrze hlavní proces lze pomocí funkce `kill` výpočet výkonného procesu kdykoliv násilně ukončit. Dalším důvodem pak je zachování funkčního serveru i v případě, že dojde k fatální chybě a výkonný proces je ukončen operačním systémem.

Při spuštění nástroje DIVINE jako server se provedou kroky vedoucí k démonizaci. To je stav, kdy program sice běží pod uživatelem, který ho spustil, ovšem uživatel nemusí být fyzicky připojen k danému stroji. V tomto stavu server vyčkává, dokud nepřijde nějaké spojení. Během čekání hlavní proces nespotřebovává žádný procesorový čas, pouze malé množství paměti potřebné k běhu. Stejná situace – žádná spotřeba procesorového času a pouze málo paměti – je i v případě, že hlavní proces dozoruje běh výkonného procesu.

Výkonný proces poskytuje distribuovanému algoritmu možnost poslat zprávu nějakému jinému procesu, přijmout zprávu od specifikovaného procesu a poslat zprávu všem procesům. Výkonný proces umožňuje mít víc kanálů mezi dvěma servery, aby bylo možné paralelně přistupovat ke komunikačnímu rozhraní . Přijímat zprávy lze ze všech kanálů, či lze specifikovat jeden kanál, na kterém bude vlákno výkonného procesu přijímat zprávy.

Je pravidlem, že výkonné procesy jsou navzájem propojeny stejným počtem kanálů, přičemž primární účel kanálů je přepravovat data distribuovaného algoritmu. Mimo to je mezi každým výkonným procesem udržováno jedno řídicí spojení. Výkonný proces je po čas výpočtu spojen s klientem jedním řídicím spojením, žádné datové kanály mezi nimi nejsou otevřeny.

Pro zachování sémantiky výstupních operací s MPI je server koncipován tak, aby zachytával jakékoliv pokusy o zápis na výstup a přeposílal je klientovi, který je zobrazí. Toho je ve výkonném procesu dosaženo přesměrováním standardního výstupu a standardního chybového výstupu a nastartováním dvou dalších vláken – jednp pro každý výstup – které se starají o samotné přeposílání. Více je popsáno v podkapitole 3.3.

Pro zjednodušení ladění chyb při vytváření nové implementace komunikační vrstvy, kdy se server fyzicky nachází na jiném stroji než spuštěný klient, jsem přidal do serveru možnost logovat jednotlivé prováděné operace. V demonstračním programu se jedná především o záznamy vstupů do funkcí, které obsluhují reakce na příkazy, a záznamy problematických situací (popsáno později).

3.2.2 Klient

Klient slouží k ovládání výpočetních serverů a nikterak se přímo nepodílí na výpočtu. Mezi základní funkce patří spuštění démonů na určených strojích, jejich ukončení, restart, dotázání se na aktuální stav (zda je server volný či zda na něm probíhá výpočet) a pak samozřejmě spuštění samotného výpočtu.

Spuštění démonů na určených strojích je aktuálně provedeno pomocí `ssh` připojení, po kterém následuje spuštění stejného programu jako je klient, ale v režimu serveru. Toto řešení má několik problémů, je například vázané na operační systém GNU/Linux z důvodu nalezení cesty ke spuštěnému klientovi.

Spuštění samotného výpočtu probíhá v několika krocích. Nejprve je potřeba ověřit, zda na všech strojích běží server a zda je dostupný – tj. neběží na něm aktuálně výpočet. Po vytvoření spojení ke každému stroji klient postupně požádá všechny servery, aby se spojily s ostatními a otevřely požadovaný počet kanálů. Následně klient může, pokud je to potřeba, doručit každému serveru blok dat, například obsah souboru. Poté nastává poslední fáze, která spočívá v přenesení argumentů z příkazové řádky a spuštění samotného výpočtu.

V průběhu výpočtu klient vyčkává na příchozí zprávy, které mohou být několika druhů. Mohou to být zachycené výstupy, které klient vypíše do správného výstupního proudu. Pokud klient obdrží datovou zprávu, přečte ji a zahodí. V neposlední řadě se může jednat o zprávu, že ten který server dokončil výpočet.

Pro spuštění poslední fáze, rozpojení sítě, je potřeba, aby všechny servery zaslaly zprávu o ukončení výpočtu. Pak klient rozpustí síť serverů, což zapříčiní ukončení výkonných procesů a hlavní procesy se uvedou do původního stavu. Následně i klient ukončí svoji činnost.

3.3 Protokol

Pro komunikaci jsem zavedl jednoduchý protokol postavený na zprávách. Zprávy se posílají v binárním formátu, takže se vyžaduje, aby všechny servery i klient měli stejnou endianitu. To může být chápáno jako případné problematické místo, ovšem neočekávám, že bude nástroj DIVINE spouštěn na hybridních výpočetních clusterech. K binárnímu formátu jsem přistoupil také kvůli rychlosti serializace.

Dále v popisu protokolu mluvím o seznamu strojů. Jedná se o seznam, který obsahuje síťové adresy strojů v lokální síti a který zároveň definuje stroje, které se budou podílet na distribuovaném výpočtu. Při adresaci není uvedené číslo portu, neboť to se nastavuje při startu klienta a musí být stejné s číslem portu, na kterých poslouchají servery uvedené v seznamu strojů.

3.3.1 Formát zpráv

Každá zpráva má hlavičku proměnlivé délky, která obsahuje následující informace:

- kategorie
- počet fragmentů
- rank odesílatele

- rank příjemce
- štítek
- velikosti fragmentů (více políček)

Kategorie
Slouží k rozeznání typů zpráv. Jinou hodnotu mají řídicí zprávy, které jsou součástí protokolu a jsou obsluhovány serverem nebo klientem, další hodnotu mají zprávy přeposílající obsah na standardní výstup a standardní chybový výstup a konečně datové zprávy, za jejichž zpracování je zodpovědný distribuovaný algoritmus. Políčko *kategorie* je 8bitové neznaménkové číslo – typ `uint8_t`.

Počet fragmentů
Udává kolik datových fragmentů má zpráva. Toto políčko je 8bitové neznaménkové číslo – typ `uint8_t`.

Rank odesílatele
Číslo serveru v rámci ustanovené sítě. Toto políčko je 8bitové neznaménkové číslo – typ `uint8_t`.

Rank příjemce
Číslo serveru v rámci ustanovené sítě. Obsahuje buď konkrétní číslo, nebo konstantu `255`, která značí, že zpráva byla zaslána hromadně. Toto políčko je 8bitové neznaménkové číslo – typ `uint8_t`.

Štítek
Slouží k určení vlastnosti předávané zprávy. V případě datové zprávy je hodnota *štítku* plně v režii distribuovaného algoritmu. Protokol je používá k označení příkazů a odpovědí a v případě přeposílání obsahu na výstup se jím rozlišuje standardní výstup od standardního chybového výstupu. Toto políčko je 32bitové neznaménkové číslo – typ `uint32_t`.

Velikosti fragmentů
Pole 32bitových čísel, které reprezentuje velikosti jednotlivých datových fragmentů zprávy. Počet fragmentů, tedy velikost pole *velikosti fragmentů*, určuje hodnota *počet fragmentů*.

Zpráva poskytuje metody pro manipulaci s hlavičkou zprávy a pro přidávání a čtení datových fragmentů do/ze zprávy. Pro omezení množství kopírovaných dat umožňuje zpráva při příjmu dat rovnou alokovat paměť pro datový fragment.

3.3.2 Seznam příkazů

Seznam příkazů, odpovědí a oznámení, které se používají v navrženém protokolu, je následující:

`OK, Refuse, Enslave, Disconnect, Peers, ConnectTo, Join, DataLine, Grouped, InitialData, Run, Start, Done, PrepareToLeave, Leave, CutRope, Error, Renegade, Status, Shutdown, ForceShutdown, ForceReset`

Každý prvek z tohoto výčtu má svoji číselnou hodnotu, která je přiřazena do *štítku* zprávy. Odpověď `OK` je akceptující odpovědí na všechny příkazy. Oproti tomu je odpověď `Refuse` zamítajícím stanoviskem.

`Enslave`
: Slouží k zotročení serveru klientem. Má tři parametry: *rank stroje*, *síťové jméno stroje* a *počet kanálů*. *Rank stroje* postupně nabývá hodnot v rozsahu 1 až počet strojů. Oproti protokolu MPI zde platí, že jsou čísla o jedno posunuté; rank 0 je vyhrazen klientovi. *Síťové jméno stroje* si server uloží do tabulky spojení, přičemž jméno je případně posléze použito pro hlášení chyb. *Počet kanálů* označuje, kolik datových kanálů bude otevřeno mezi každou dvojcí strojů.

`Disconnect`
: Použije klient v případě, že hodlá ukončit spojení se serverem.

`Peers`
: Slouží k zahájení vytváření sítě. Má jeden parametr: *velikost světa*. Ten označuje počet strojů ve skupině. Příkaz zasílá klient serverům.

`ConnectTo`
: Slouží jako žádost klienta o vytvoření spojení mezi serverem **A** a serverem **B**. Příkaz má tři parametry: *rank stroje*, *síťové jméno stroje* a *adresu*. První dva parametry jsou stejné jako u předchozího příkazu. *Adresu* použije server **A** pro připojení k serveru **B**.

`Join`
: Použije server **A** pro otevření řídicího spojení k serveru **B**. Má dva parametry: *rank* serveru **A** a *síťové jméno* serveru **A**.

`DataLine`
: Použije server **A** pro otevření datového komunikačního kanálu k serveru **B**. Má dva parametry: *rank* serveru **A** a *číslo kanálu*.

`Grouped`
: Posílá klient serverům v okamžiku dokončení propojování serverů.

`InitialData`
: Klient příkaz posílá spolu s blokem počátečních dat všem serverům. Volitelný příkaz.

`Run`
: Příkazem pobídne klient servery, aby spustili distribuovaný výpočet. Jako parametry jsou argumenty, které budou předány spouštěné funkci.

`Start`
: Slouží jako bariéra. Po přijetí server spouští distribuovaný výpočet.

`Done`
: Oznámení, které posílají servery klientovi po dokončení distribuovaného algoritmu.

`PrepareToLeave`
: Příkazem klient připraví výkonný proces na ukončení činnosti.

`CutRope`
: Oznámení, kterým dává na serveru výkonný proces najevo hlavnímu procesu, že ukončuje svoji činnost.

`Leave`
: Příkaz, jehož přijetím výkonný proces ukončí svoji činnost.

`Error`
`Renegade`
: Příkazy slouží k oznámení chyby všem propojeným strojům. Příkaz `Renegade` má jeden parametr: *adresu* stroje, kde došlo k chybě.

`Status`
: Slouží pro dotaz, v jakém stavu se server nachází, i pro odpověď serveru.

`Shutdown`
`ForceShutdown`
: Slouží k ukončení činnosti serveru. `ForceShutdown` ukončí server vždy, `Shutdown` pouze tehdy, když se server neúčastní výpočtu.

`ForceReset`
: Slouží k násilné obnově výchozího stavu serveru.

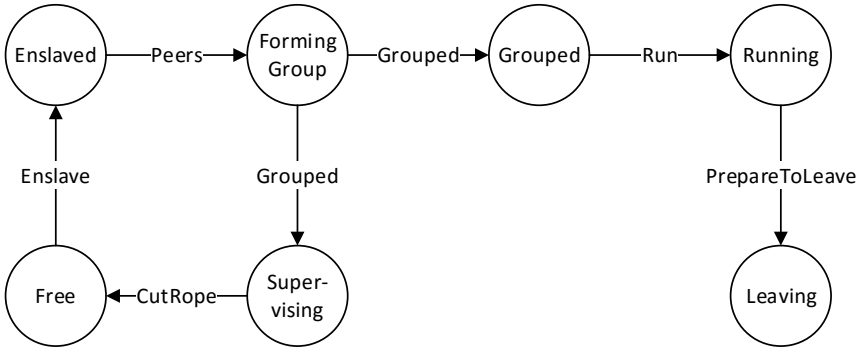
3.3.3 Stavy serveru

Server se může v průběhu svého běhu nacházet v jednom z následujících stavů:

1. *Free*. Server je k dispozici.
2. *Enslaved*. Server byl osloven klientem a podřídil se mu.
3. *FormingGroup*. Server je ve fázi vytváření spojení ke všem ostatním serverům ve skupině.
4. *Grouped*. Všechny servery jsou navzájem spojeny a došlo k vytvoření výkonného procesu.

5. *Supervising*. Hlavní proces začal dohlížet na výkonný proces.
6. *Running*. Výkonný server spustil distribuovaný algoritmus.
7. *Leaving*. Výkonný server započal ukončující fázi.

Některé stavy jsou vyhrazeny pouze pro hlavní proces (*Free*, *Enslaved*, *FormingGroup*, *Supervising*). Do zbývajících stavů se může dostat pouze výkonné vlákno. Pokud se klient dotáže serveru na to, v jakém je stavu, jako odpověď dostane jeden ze stavů hlavního procesu. Zjistit stav výkonného procesu není možné.



Obrázek 3.1: Přechodový diagram stavů

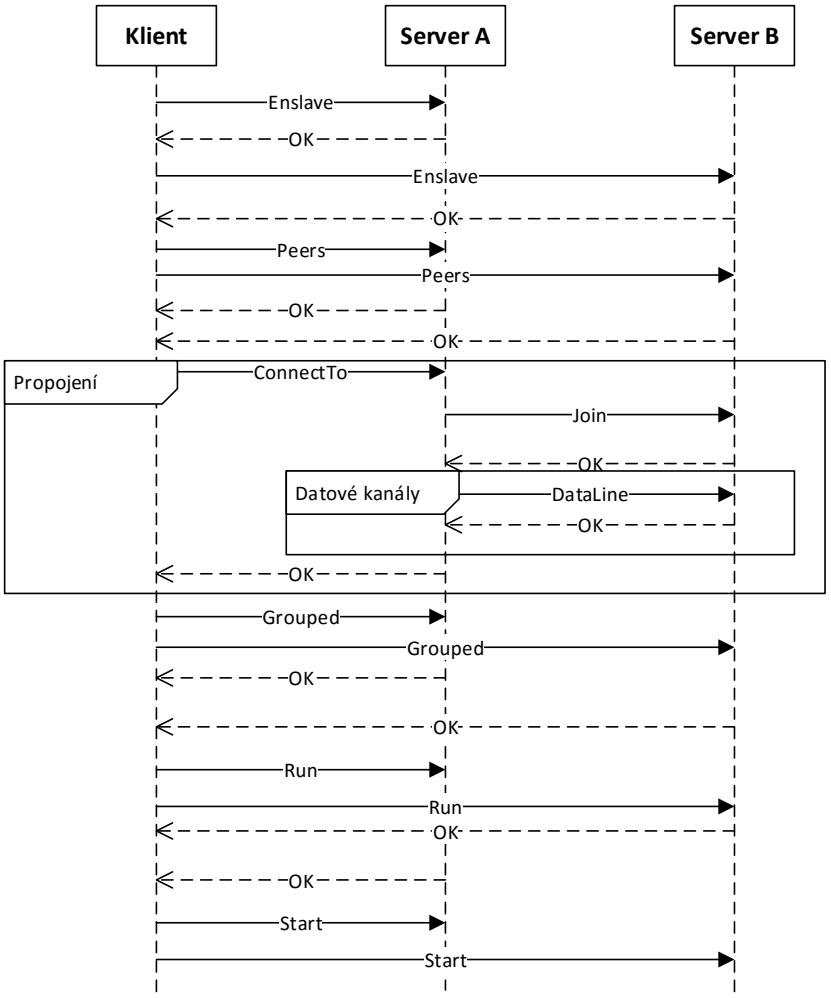
Server může měnit svůj stav na základě příkazu od klienta. Na základě jakého příkazu, případně události, server změní svůj stav je popsáno na 3.1. Dále zde platí to, že téměř všechny příkazy od klienta může server akceptovat, pouze pokud je ve správném stavu. Přechody mezi jednotlivými stavy jsou tak primárně kontrolní mechanizmus.

3.3.4 Ustanovení sítě

Jak již bylo lehce zmíněno, ustanovení sítě probíhá ve třech fázích: navázání spojení, propojení a spuštění. Na 3.2 je zakreslen průběh úspěšného ustanovení sítě. Jednotlivé fáze jsou pak rozepsány níže.

Navázání spojení

Klient postupně otevírá spojení na každý stroj ze seznamu strojů. Po každém úspěšném připojení klient zašle serveru příkaz `Enslave`, na nějž může server odpovědět třemi způsoby. Pokud je ve stavu *Free*, může příkaz akceptovat, čímž sám sebe přepne do stavu *Enslaved*. Nebo může server zamítnout zotročení v případě, že se nachází v jiném stavu než *Free*. Pokud server odpoví jinak, je odpověď nahlášena jako chybná a řeší se jako chybový stav.



Obrázek 3.2: Ustanovení sítě

Pokud se klient není schopný k některému stroji připojit, případně dostane zamítavou odpověď, rozešle všem již připojeným serverům zprávu `Disconnect`, počká na odpověď a ukončí všechna spojení. Zpráva `Disconnect` způsobí, že server pošle akceptující odpověď, přejde zpět ze stavu *Enslaved* do stavu *Free* a smaže všechny nabyté znalosti. V opačném případě – podařilo se vytvořit spojení se všemi stroji – pokračuje klient druhou fází: propojením.

Propojení

V této fázi klient rozešle všem zotročeným serverům příkaz `Peers` spolu s jedním parametrem velikosti světa. Přijetím tohoto příkazu se server přepne ze stavu *Enslaved* do *FormingGroup* a zapamatuje si velikost světa.

Server neakceptuje příkaz `Peers` pouze v případě, že se nenachází ve stavu *Enslaved*. Tato situace je řešena jako chybový stav, neboť k ní může dojít pouze chybou v protokolu samotném.

Po obeznámení všech serverů s velikostí světa posílá klient postupně jednotlivým serverům žádosti o spojení s dalšími servery, což provádí příkazem `ConnectTo`. Na základě žádosti požádaný server **A** postupně pomocí příkazů `Join` a `DataLine` ustanoví spojení s cílovým serverem **B**. Počet zaslání příkazů `DataLine` zavisí na počtu požadovaných otevřených datových kanálů. Pokud se podaří navázat všechna spojení, pošle server **A** klienovi odpověď `OK`, v opačném případě odpoví `Refuse`.

Pokud klient obdrží od některého serveru odpověď `Refuse`, řeší se nastalá situace jako chybový stav. Jinak klient zašle serverům příkaz `Grouped`, který způsobí, že na každém serveru hlavní proces vytvoří výkonný proces. Možnost přijímat příchozí spojení zůstane pouze hlavnímu procesu, naopak výkonný proces dostane na starost všechny již otevřená spojení. Výkonný a hlavní proces spolu zůstanou v kontaktu pomocí anonymního páru socketů vytvořených funkcí `socketpair`. Hlavní proces se přepne do stavu *Supervising*, zatímco výkonný proces se přepne do stavu *Grouped*. Od tohoto okamžiku klient komunikuje pouze s výkonným procesem.

Spuštění

Poslední fází k započetí distribuovaného algoritmu je spuštění. Před samotným spuštění může klient rozeslat počáteční data všem výkonným procesům pomocí příkazu `InitialData`. Po něm již následuje zaslání příkazu `Run` spolu s parametry příkazové řádky, kterým se výkonný proces přepne ze stavu *Grouped* do stavu *Running*.

Samotný start algoritmu v sobě implementuje bariéru, kdy je až příkazem `Start` od klienta každý výkonný proces zpraven o skutečném odstartování. Tento krok je v protokolu použit, aby nedocházelo na jednom stroji k započetí výpočtu, když se klient ještě stará o rozeslání dat zbylým strojům.

Před samotným startem distribuovaného algoritmu se navíc spustí dvě vlákna, která mají na starost přesměrování standardního výstupu a standardního chybového výstupu na klienta. Klient se tak od tohoto okamžiku stará pouze o zobrazování přeposlaných výstupů a případné řešení chybových stavů.

3.3.5 Rozpuštění sítě

Rozpuštění sítě započne v okamžiku, kdy první stroj pošle oznámení `Done`, kterým informuje klienta o dokončení výpočtu. Až klient obdrží tato oznámení od všech strojů, přejde k samotnému ukončení. To zahrnuje rozeslání příkazu `PrepareToLeave` od klienta všem serverům, který způsobí, že se klient přepne ze stavu *Running* do stavu *Leaving*. V tomto stavu přestává server hlásit případné výpadky spojení.

Jakmile dostane klient odpověď od všech výkonných procesů, že přešly do stavu *Leaving*, rozešle jim příkaz `Leave`. Ten způsobí, že každý výkonný proces pošle oznámení `CutRope` svému hlavnímu procesu, po jehož přijetí se hlavní proces na každém serveru přepne ze stavu *Supervising* do stavu *Free*. Souběžně s tím dojde na straně každého výkonného procesu k uzavření všech spojení a k ukončení činnosti.

Na komunikaci mezi výkonným a dohlížejícím procesem je klíčové, aby si výkonný proces počkal na potvrzení přijetí oznámení `CutRope` od hlavního procesu. Vynecháním potvrzení totiž může docházet k situaci, kdy klient již ukončil svoji činnost, ale hlavní proces je stále ve stavu *Supervising*, což může mít za následek, že další spuštění výpočtu za použití stejných serverů nebude možné provést okamžitě, ale až za nějaký časový moment.

3.3.6 Ostatní příkazy

Klient může operovat s dalšími příkazy: `Status`, `Shutdown` a `ForceShutdown`.

Pomocí dotazu `Status` může klient zjistit, v jakém stavu se nachází server. Běžně je pro zjištění stavu vytvořeno nové spojení na server. Ten po obdržení dotazu na svůj stav pošle zprávu s textovým popisem svého stavu. Server se nikterak nestará o nové spojení, které tím pádem zaniká. Server na dotaz na svůj stav odpoví vždy, ať už je v jakémkoliv stavu. Protože se ale klient dotazuje pouze hlavního procesu, může dostat jako odpověď pouze tyto stavy: *Free*, *Enslaved*, *FormingGroup* a *Supervising*.

Příkaz `Shutdown` je zdvořilou žádostí o ukončení běhu serveru. Serveru tuto žádost akceptuje pouze v případě, že se nachází ve stavu *Free*.

Příkaz `ForceShutdown` je silnější variantou předchozího příkazu, která zaručí, že server ukončí svoji činnost nezávisle na stavu, ve kterém se nachází. Navázaná spojení jsou ukončena bez zaslání jakékoliv zprávy a v případě, že je nastartován výkonný proces, je násilně ukončen.

Poslední příkaz – `ForceReset` – slouží k tomu, aby hlavní proces přešel do stavu *Free*, ať už byl předtím v kterémkoliv jiném stavu, což zahrnuje také násilné ukončení výkonného procesu, pokud je spuštěn. Tento příkaz kromě explicitního vyžádání od uživatele použije klient v případě, že obdržel signál, který by způsobil ukočení klienta. Zejména se jedná o signály `SIGALRM`, `SIGINT`[2] a `SIGTERM`.

[2]Tento signál je na unixových systémech většinou generován prostřednictvím klávesové

3.3.7 Řešení chyb a problémových stavů

Mnou navržený protokol je připraven řešit některé problémové situace, které mohou nastat. Problémové situace lze rozdělit do několika okruhů: selhání síťových komponent, vnitřní chyba protokolu a problém s distribuovaným algoritmem.

Mnohé z potenciálně problematických operací jsou ošetřeny nastavením maximálního časového úseku, po který mohou operace blokovat provádění. Pokud ani po uplynutí časového úseku nebylo možné operaci dokončit, dojde ve většině případů k vyhození výjimky `WouldBlock`.

Jakákoliv vyhozená výjimka je, pokud toto chování distribuovaný algoritmus neupraví, zachycena v nejvrchnější funkci, je reportována uživateli – klient vypíše chybovou hlášku na obrazovku, server ji zaznamená do logů – a program je ukončen. Odlišné chování má akorát hlavní serverový proces, který sice zachycenou výjimku zaznamená do logů, ovšem neukončí svoji činnost, ale uvolní veškeré alokované zdroje, případně násilně ukončí běh výkonného procesu, a přejde do stavu *Free*. Tento mechanizmus zpracování výjimek jsem zvolil, aby při nastalých problémech nedocházelo k uváznutí, a naopaka aby byl uživatel zpraven o nastalé chybě.

Za chyby se v okruhu selhání síťových komponent považuje cokoliv od zamítnutí připojení až po rozpojení sítě během výpočtu. V mé implementaci protokolu většinu těchto problémů řeší objektová nadstavba nad POSIX rozhraním BSD socketů.

Problémy vzniklé během vytváření spojení mezi dvěma stroji mohou mít několik příčin. Pokud se nepodaří přeložit název cílového stroje na IP adresu, ať již z důvodu nedostupnosti stroje, či protože je v názvu překlep, dojde k vyhození výjimky s relevantním popiskem. Jiná výjimka může být vyhozena, pokud se do určitého časového okamžiku nepodaří spojení navázat.

Po úspěšném navázání spojení je každý nově otevřený socket poznačen jako blokující[3] a zároveň mu je nastaveno, jak nejdéle může trvat operace nad socketem. Časový limit v tomto případě ohraničuje především operace čtení a zápisu do socketu. Operace zápisu se může zdržet například proto, že cílový stroj si nevyzvedává příchozí zprávy a vyrovnávací paměť operačního systému pro příchozí data je již plná. U operace čtení může naopak dojít k tomu, že zdrojový stroj nepošle žádná data. Tímto se řeší především situace, kdy dojde k uváznutí na některém serveru z důvodu chybného distribuovaného algoritmu. Dalším problémovým místem pak může být ztráta spojení mezi dvěma stroji v průběhu výpočtu. Tato situace je opět zachycena objektovou nadstavbou, jejíž reakce je vyhození výjimky `ConnectionAbortedException`.

Pro eliminaci chyb v samotném protokolu jsem přistoupil jednak k použití

zkratky `Ctrl+C`.

[3]Každý získaný socket je blokující. Explicitně tuto informaci uvádím, protože během připojení jsou sockety z důvodu prevence uváznutí označeny jako neblokující a ke změně na blokující dojde až po vytvoření spojení.

stavů u serverových procesů, jednak k posílání odpovědí na každý příkaz – vyjma příkazů `ForceShutdown` a `ForceReset`. Server odmítne vykonat příkaz, pokud pro jeho splnění nejsou vhodné podmínky, například protože se jeho procesy nachází ve špatných stavech.

Reakce na odmítnutí příkazu se může lišit dle závažnosti. Některá odmítnutí nemusí být závažná a mohou být ignorována či pouze ohlášena uživateli (forma závisí na tom, zda se jedná o klienta či server), některá ovšem mohou způsobit vyhození výjimky. Většina zamítnutí ovšem vyústí ve vyhození výjimky.

Nevhodný návrh distribuovaného algoritmu spolu s chybnou implementací mohou vyústit v takové uváznutí, kdy některý výkonný proces nebude pracovat s otevřenými spojeními, což může i v případě ukončení klienta vést k tomu, že tento výkonný proces nebude ani po násilném rozpuštění sítě schopen zastavit výpočet. Z toho důvodu je v rámci klienta použito zachytávání některých signálů (zejména `SIGINT` a `SIGTERM`), které způsobí, že klient zašle všem uváznutým serverům (konkrétně hlavním procesům) příkaz `ForceReset`.

3.3.8 Bezpečnost

Protokol jako takový byl koncipován na provoz v bezpečném prostředí, takže nepodporuje žádnou úroveň zabezpečení. Ze základních prvků zabezpečení zde chybí hlavně autentizace klienta vůči serveru a kontrola integrity dat.

Žádná metoda zabezpečení nebyla nasazena ze dvou důvodů. První z nich byl již vyřčen – očekávám, že nástroj DIVINE bude provozován v zabezpečené vnitřní síti. Druhým důvodem je velký důraz na rychlost. Ačkoliv by nasazení autentizace sice nešlo proti druhému důvodu, implementace integrity dat například pomocí SSL by již mohla způsobit snížení rychlosti toku dat. Protože však záměr práce nespočívá v nasazení zabezpečovacích prvků, je dle mého soudu zbytečné tyto mechanizmy do protokolu implementovat.

3.4 Rozhraní pro distribuované procházení grafu

Při návrhu rozhraní jsem vycházel převážně z požadavků nástroje DIVINE. Všechny aktuálně implementované distribuované algoritmy využívají z knihovny MPI pouze několik málo funkcí pro komunikaci – `MPI_Send` / `MPI_Isend`, `MPI_Probe` / `MPI_Iprobe` a `MPI_Recv` / `MPI_Irecv`. Ačkoliv některé algoritmy navíc zasílají zprávu všem spolupracujícím instancím, k tomu se ale nevyužívá funkce `MPI_Bcast`, nýbrž opakované volání funkce pro zaslání zprávy.

Dalším požadavkem bylo, aby šla komunikační vrstva použít v paralelním kontextu bez nutnosti řešit zamykání uvnitř algoritmu a tím lépe využít paralelizmu. Přestože jsem neměl k dispozici kompletní požadavky na rozhraní z důvodu souběžně vyvíjené nové verze nástroje DIVINE, snažil jsem se co nejvíc vyhovět požadavkům, které jsem dostal. Způsob komunikace – zasílání zpráv – zůstává nadále platný i pro novou verzi.

3.4.1 Zprávy

Zprávy jsou v rozhraní reprezentovány dvěma třídami. Třída `InputMessage` reprezentuje příchozí zprávu, třída `OutputMessage` pak odchozí zprávu. Důvod pro dvě odlišné třídy je daný jiným očekávaným chováním každé z nich, především co se práce s pamětí týká.

Do odchozí zprávy je možné přidat data buď pomocí metody `add` nebo pomocí operátoru `<<`. Druhý způsob umožňuje připojit ke zprávě i třídu `std::string` případně `std::vector`. Pro data připojené pomocí operátoru `<<` navíc platí, že typ dat musí splňovat koncept `TriviallyCopyable`[4].

Připojení konstantních dat způsobí vytvoření jejich kopie. K tomuto kroku jsem přistoupil, protože použitá funkce z rozhraní BSD socketů (`sendmsg`) neumožňuje předat ukazatel na konstantní data. Vzhledem ke stávajícímu způsobu práce s pamětí v nástroji DIVINE očekávám, že i nadále bude výskyt konstantních dat sporadický. Co je ovšem důležitější, je platnost nekonstantních dat, kdy tato data musí být validní dokud nedojde k odeslání zprávy.

Příchozí zpráva také poskytuje více nástrojů pro práci s pamětí. Jednou možností je použít operátor `>>` a nastavit tak, do které proměnné se má nastavit fragment dat. V případě použití této možnosti je potřeba zaručit, aby nastavný počet a velikosti fragmentů odpovídaly fragmentům skutečně přijatým; v opačném případě dojde k vyhození výjimky.

Druhou možností je nastavit při příjmu vlastní funkci, která bude alokovat paměť postupně pro každý datový fragment, čímž se programátor stává zodpovědný za to, že alokovaná paměť bude korektně uvolněna.

3.4.2 Služby komunikačního rozhraní

Ačkoliv lze chápat služby komunikačního rozhraní jako funkce, případně jako statické metody, implementoval jsem je z důvodů přehlednosti jako metody třídy `Daemon`, případně předka `Communicator`. Tato třída má vytvořenou jednu globální instanci, která je dostupná krze statickou metodu `instance`.

Rozhraní poskytuje tyto několik metod, které by bylo možné rozdělit na dvě kategorie: komunikační a servisní. Komunikační, jak již název evokuje, slouží k zasílání a přijímání zpráv. Servisní sdružují metody, pomocí kterých lze získávat informace o prostředí.

Servisní metody

`int Communicator::rank() const;`
: Metoda vrací rank procesu.

`bool Daemon::master() const;`
: Metoda vrací `true`, pokud je rank procesu roven 1.

[4] http://en.cppreference.com/w/cpp/concept/TriviallyCopyable

`int Communicator::worldSize() const;`
Metoda vrací počet procesů ve skupině.

`int Communicator::channels() const;`
Metoda vrací počet dostupných datových kanálů mezi procesy.

`const std::string &Communicator::name() const;`
Metoda vrací název stroje, na kterém běží proces.

`char *Daemon::data() const;`
Metoda vrací ukazatel na počáteční data.

`size_t Daemon::dataSize() const;`
Metoda vrací velikost počátečních dat v bytech.

`void Daemon::exit(int returnCode);`
Metoda zahlásí vedoucímu procesu ukončení činnosti a následně program. Parametr *returnCode* slouží jako návratový kód procesu. Slouží jako náhrada za knihovní funkci `exit`, která ovšem neukončí korektně komunikaci s ostatním procesy.

Metoda nesmí být volána v průběhu provádění metody `probe`, pokud je metoda `probe` volána nad hlavním komunikačním kanálem, neboť by mohlo dojít k uváznutí.

Komunikační metody

Většina komunikačních metod může při problémech v síti vyhazovat různé výjimky. Pro všechny výjimky platí, že jsou třídami odvezenými ze třídy `brick::net::NetException`, která je sama odvozena ze třídy `std::exception`.

`bool Daemon::sendTo(int rank, OutputMessage &msg, ChannelID chID);`
`bool Daemon::sendTo(Channel channel, OutputMessage &msg);`

Základní metody sloužící k zaslání zprávy *msg*. První z uvedených funkcí přeloží *rank* a *chID* na kanál *channel*, kterým se pošle zpráva, a zavolá se druhá uvedená metoda. Pokud není parametr *chID* uveden, použije se řídicí kanál. Sémantika odeslání je vzhledem k použití systémového volání `sendmsg` blokující bafrovaná operace.

Funkce vrací `true`, pokud byla operace úspěšně provedena. Vrací `false` v případě chybného parametru *rank* nebo *chID* – neexistující rank nebo neexistující kanál. Metody mohou vyhodit tyto výjimky:

- `WouldBlock` – Použitý socket má příznak neblokující a očekávaný přenos by způsobil blokování, případně došlo k dosažení časového limitu odesílací operace. Proměnná `errno` má hodnotu `EAGAIN` nebo `EWOULDBLOCK`.

- `ConnectionAbortedException` – Došlo k chybě ve spojení. Proměnná `errno` má hodnotu `EPIPE`, `ENOTCONN`, nebo `ECONNREFUSED`.
- `SystemException` – Došlo k chybě při přenosu a hodnota proměnné `errno` je odlišná od již uvedených.
- `DataTransferException` – Došlo sice k úspěšnému přenosu, ale byla přenesena pouze část zprávy.

```
bool Daemon::receive(int rank, InputMessage &msg, ChannelID
chID);
bool Daemon::receive(Channel channel, InputMessage &msg);
```

Základní metody sloužící k příjmu zprávy *msg* z konkrétního komunikačního kanálu. Obdobně jako u metod u první `sendTo` zde platí, že parametry *rank* a *chID* se přeloží na kanál *channel* a použije se druhá uvedená metoda `receive`. Pokud není parametr *chID* uveden, použije se řídicí kanál. Sémantika přijetí zprávy je blokující nebafrovaná operace.

Funkce vrací `true`, pokud byla operace úspěšně provedena. Vrací `false` v případě chybného parametru *rank* nebo *chID* – neexistující rank nebo neexistující kanál. Metody mohou vyhodit tyto výjimky:

- `WouldBlock` – Použitý socket má příznak neblokující a očekávaný přenos by způsobil blokování, případně došlo k dosažení časového limitu přijímací operace. Proměnná `errno` má hodnotu `EAGAIN` nebo `EWOULDBLOCK`.
- `ConnectionAbortedException` – Došlo k chybě ve spojení. Proměnná `errno` má hodnotu `ENOTCONN`, nebo `ECONNREFUSED`.
- `SystemException` – Došlo k chybě při přenosu a hodnota proměnné `errno` je odlišná od již uvedených.
- `DataTransferException` – Došlo sice k úspěšnému přenosu, ale byla přijata pouze část zprávy.

```
bool Daemon::sendAll(OutputMessage &msg, ChannelID chID);
```

Metoda sloužící k zaslání dané zprávy *msg* všem procesům ve skupině pomocí kanálu *chID*. Pokud se neuvede parametr *chID*, bude jako komunikační kanál zvolen řídicí kanál.

Aktuální implementace této metody je taková, že dojde k vytvoření *N - 1* vláken, které každé pošle zprávu jednomu procesu. Po skončení odesílání se zkontroluje, zda se podařilo zprávu poslat všem. Pokud ano,vrátí metoda hodnotu `true`, jinak `false`. Metoda nevyhazuje žádné výjimky.

Vzhledem k tomu, že je metoda málo používaná, zvolil jsem řešení s paralelním odesíláním. Pokud by v budoucnu začala metoda být více používaná, bylo by vhodné změnit implementaci odesílání z paralelní na hyperkubickou [31].

`int Daemon::probe(Ap applicator, ChannelID chID, int timeout);`

Metoda, která zkouší přijímat příchozí spojení od všech procesů na kanálu *chID*. Na kanálech, ze kterých lze přijímat zprávy, se postupně volá funkce *applicator*.

Funkce *applicator* může mít dvojí signaturu:

- `void applicator(Channel channel);` – Základní varianta funkce *applicator* může skrze parametr *channel* zpracovat příchozí zprávu, případně také skrze stejný kanál odpovědět.
- `bool applicator(Channel channel);` – Rozšířená varianta funkce *applicator* může navíc v návratové hodnotě sdělit, jestli se má pokračovat v aplikování funkce *applicator* na další připravené kanály. Pro pokračování je třeba vrátit `true`, pro ukončení zpracování `false`.

Parametr *timeout* udává ve vteřinách, jak dlouho má metoda `probe` čekat. Pokud je *timeout* roven *0*, nebude `probe` čekat vůbec, naopak pokud bude hodnota parametru *-1*, bude čekat, dokud nebude některé spojení připraveno na příjem zprávy. Implicitní hodnota parametru *timeout* je *-1*.

Metoda `probe` vrací číselnou hodnotu kolikrát byla na příchozí spojení zavolána funkce *applicator*. Počet zpracovaných příchozích spojení se ale může lišit, neboť metoda `probe` zpracovává veškeré zprávy, ale jenom zprávy kategorie `Data` předává ke zpracování funkci *applicator*.

Každý proces by měl v jednom vlákně pomocí metody `probe` pravidelně kontrolovat řídicí kanály, neboť uvnitř metody dochází k obsluze protokolu.

Poznámka: Metoda `probe` by se měla volat periodicky v cyklu, neboť se může vrátit aniž by zavolala funkci *applicator* – buď z důvodu vypršení časového limitu, nebo z důvodu zpracování zprávy jiné kategorie než datové.

Kapitola 4

Experimentální porovnání

Nová implementace poskytuje nástroji DIVINE několik způsoby použití, z nichž jsem vybral dva relevantní scénáře. První z nich je, že bude v rámci každého procesu jedno dedikované vlákno, které bude přijímat zprávy od ostatních procesů, přičemž stejná situace je u stávající komunikační vrstvy, která používá MPI. Tento scénář použití nové implementace je v měřeních popsán jako *Shared*.

Druhý možný scénář použití nového komunikačního rozhraní je, že každý proces bude spojen s ostatními procesy tolika datovými kanály, kolik pracovních vláken bude v rámci každého procesu spuštěno.[1] Každé pracovní vlákno bude mít identifikační číslo v rámci procesu a sadu kanálů, které ho budou spojovat s vlákny stejného identifikačního čísla v jiných procesech. Vlákna tak spolu budou komunikovat jednak pomocí sdílené paměti v rámci jednoho procesu a každé vlákno může kontaktovat jedno vlákno z každého dalšího procesu, k němuž bude mít samostatný kanál. Tento scénář je v měřeních popsán jako *Dedicated*.

Tyto scénáře použití nového komunikačního rozhraní se spolu s MPI podrobí experimentálnímu porovnání a na základě časového měření zhodnotím jejich chování. Pozornost zaměřuji kromě absolutních časových hodnot na schopnost škálovat, a to jak v počtu vláken na proces, tak v počtu procesů samých.

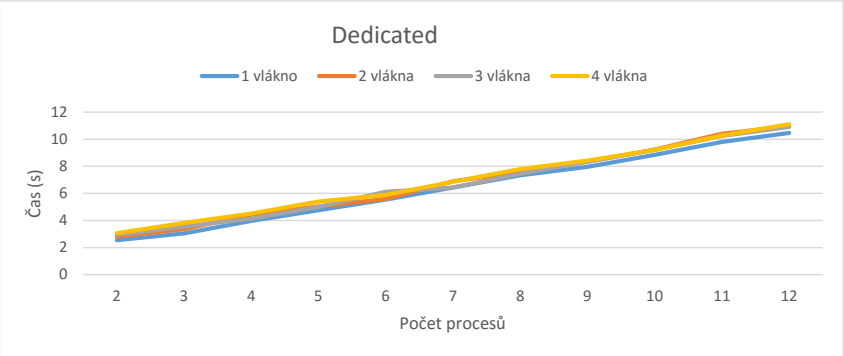
Měřeny byly celkem dva testy: test latence a test posílání zpráv. Měření probíhala na *22* strojích `pheme`. Pro každé měření byl náhodně vybrán požadovaný počet strojů, aby byla snížena možnost, že některý stroj bude pracovat pomaleji než ostatní. Každé měření bylo *10* opakováno, aby bylo sníženo riziko náhodných výkyvů v síti, a z těchto měření

[1] To mimo jiné implikuje, že všechny procesy budou mít spuštěný stejný počet vláken.

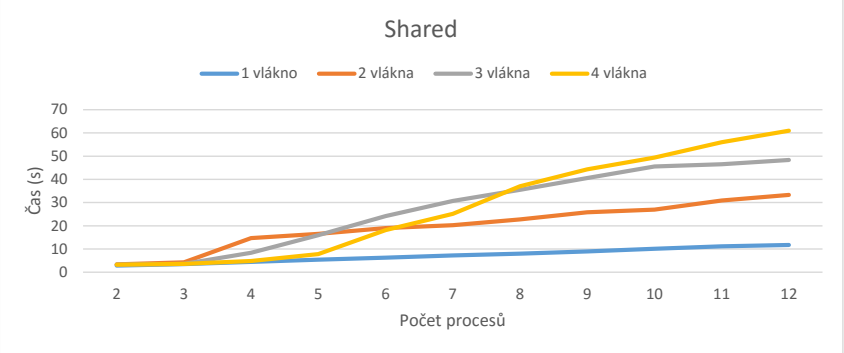
4.1 Test latence

Test latence probíhá tak, že každé vlákno postupně vygeneruje čísla od *1* až po *10 000*, a pošle je na zpracování jinému procesu, které ho obratem vrátí zpátky, akorát vynásobené *-1*. Pro každé číslo si vlákno pomocí pseudonáhodného generátoru čísel určí proces, kterého se bude dotazovat.

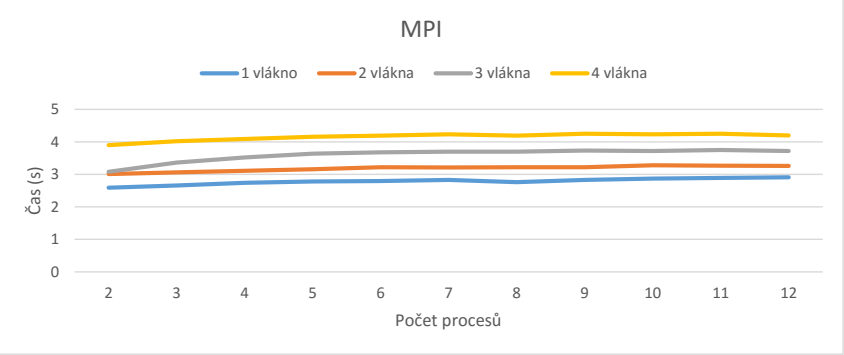
4.1.1 Porovnání dle scénářů



Obrázek 4.1: Škálování vláken při použití scénáře Dedicated

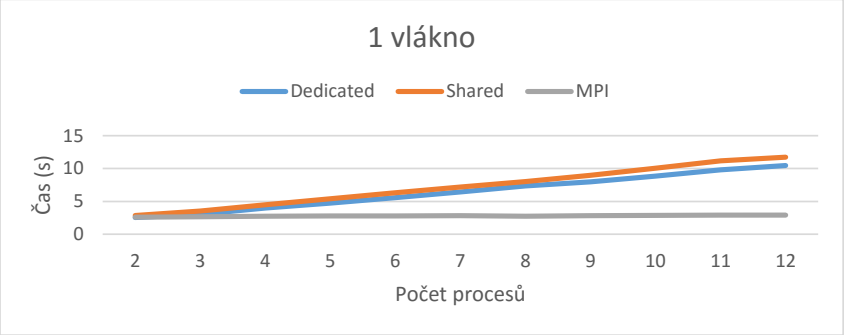


Obrázek 4.2: Škálování vláken při použití scénáře Shared

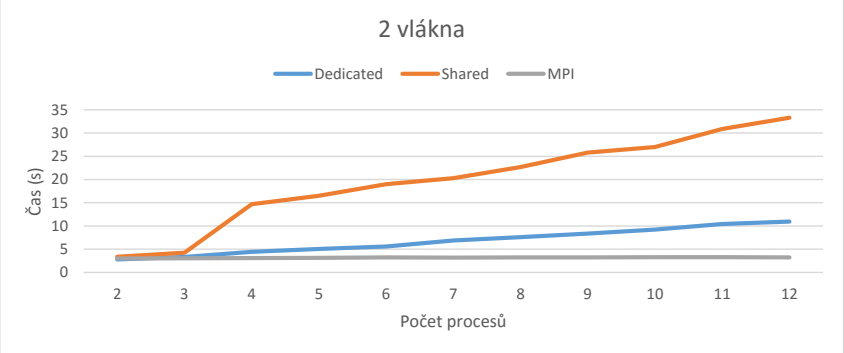


Obrázek 4.3: Škálování vláken při použití scénáře MPI

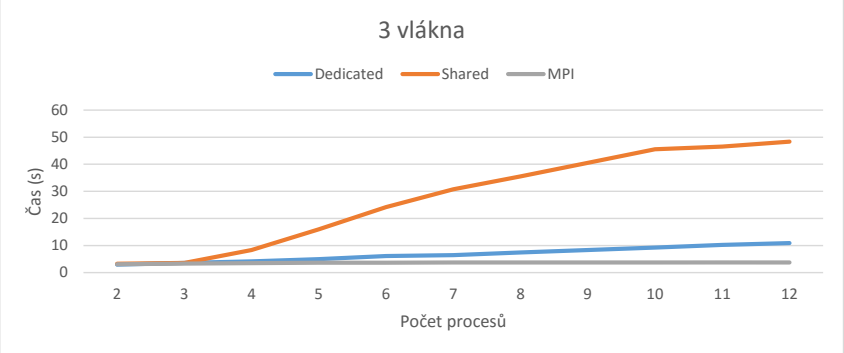
4.1.2 Porovnání dle počtu vláken



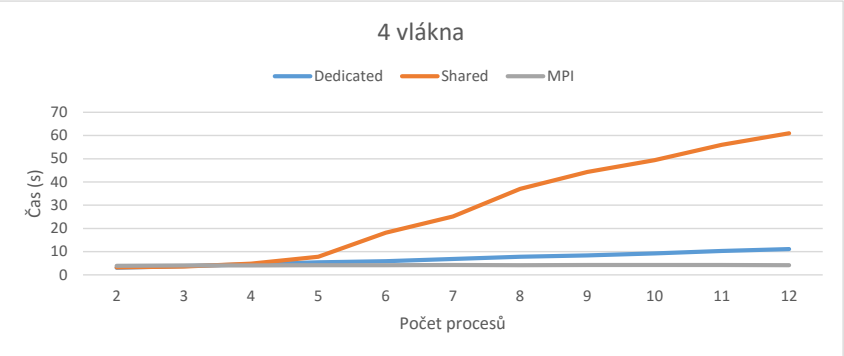
Obrázek 4.4: Porovnání scénářů při 1 vlákně



Obrázek 4.5: Porovnání scénářů při 2 vláknech



Obrázek 4.6: Porovnání scénářů při 3 vláknech

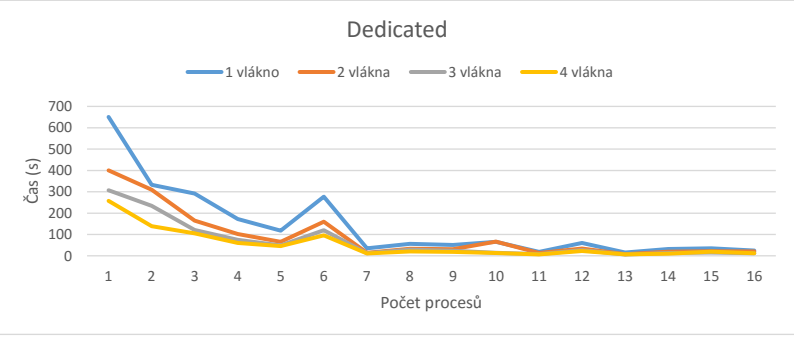


Obrázek 4.7: Porovnání scénářů při 4 vláknech

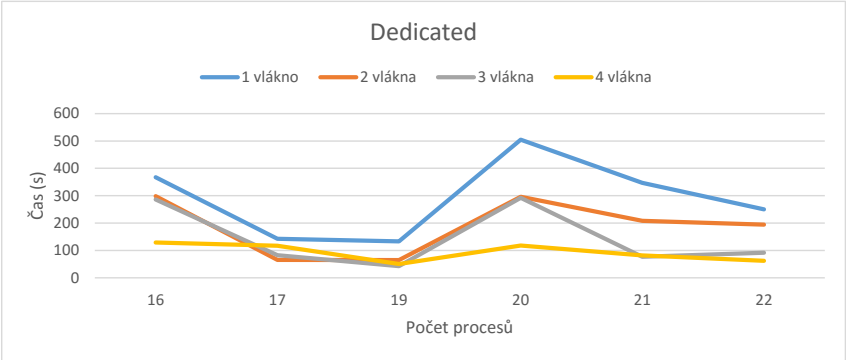
4.2 Testy škálovatelnosti

Testy škálovatelnosti simulují průchod orientovaným grafem v obdobném duchu, jakým ho prochází nástroj DIVINE. Vrchol grafu je reprezentován dvěma čísly. V jednom z procesů dojde k vytvoření počátečního vrcholu grafu s nulovými hodnotami, který je poslán na zpracování. Zpracování probíhá tak, že je postupně jedno a druhé číslo z reprezentace vrcholu navýšeno o *1* a následně odesláno k dalšímu zpracování. Každý vrchol je na základě haše jednoznačně přidělený procesu s určitým rankem. V rámci testů je ukončení výpočtu zaručeno nejvyšší možnou hodnotou obou čísel reprezentující vrchol, po jejichž dosažení nedochází dále k předávání nově vytvořených vrcholů ke zpracování. V testu je pro *1* — *16* procesů postupně generován graf o *9 000 000* vrcholech a pro *16* — *22* procesů má generovaný graf ve výsledku *100 000 000* vrcholů.

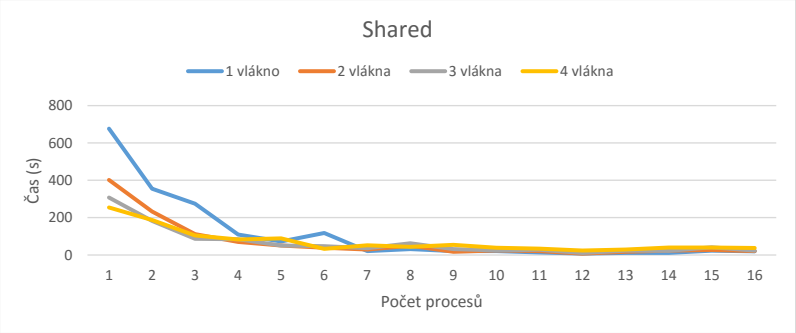
4.2.1 Porovnání dle scénářů



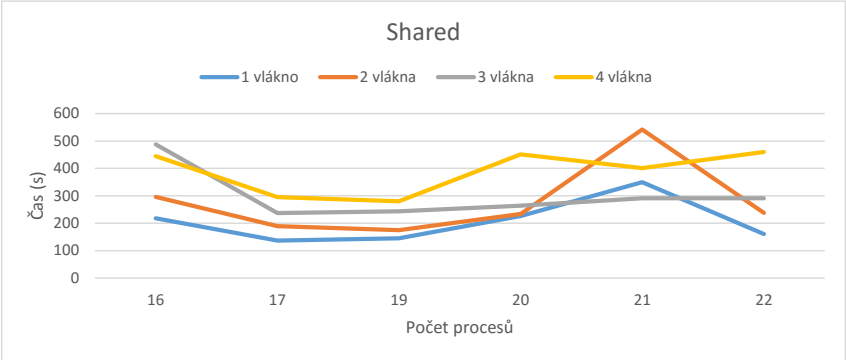
Obrázek 4.8: Škálování vláken při použití scénáře Dedicated: 1 — 16 procesů



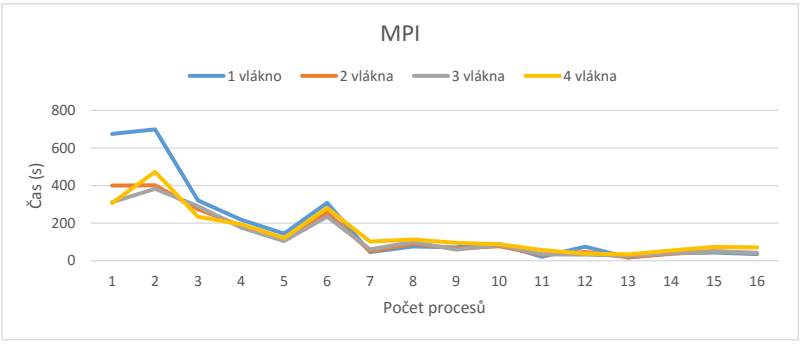
Obrázek 4.9: Škálování vláken při použití scénáře Dedicated: 16 – 22 procesů



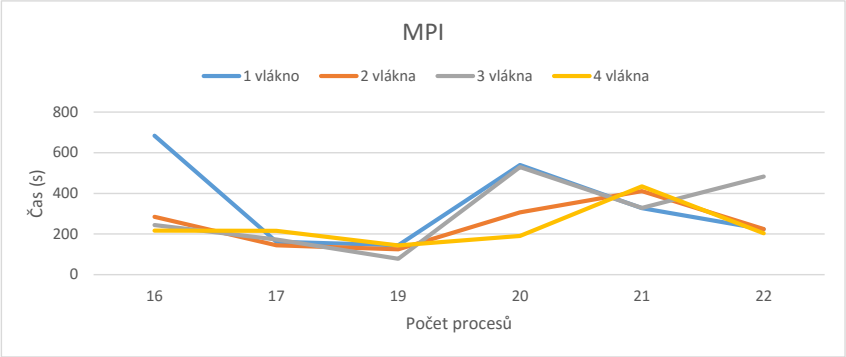
Obrázek 4.10: Škálování vláken při použití scénáře Shared: 1 — 16 procesů



Obrázek 4.11: Škálování vláken při použití scénáře Shared: 16 — 22 procesů

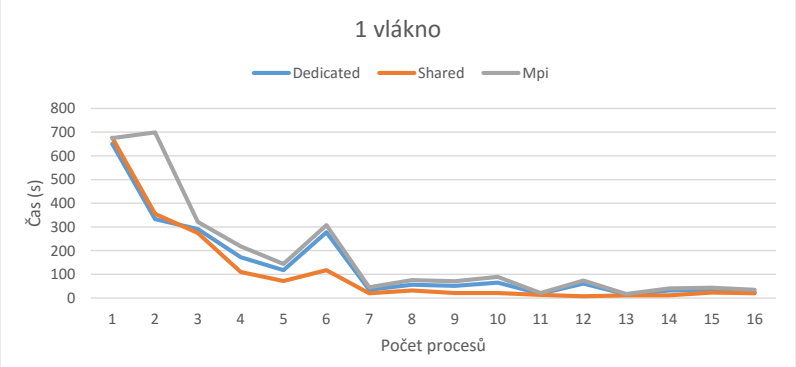


Obrázek 4.12: Škálování vláken při použití scénáře MPI: 1 — 16 procesů

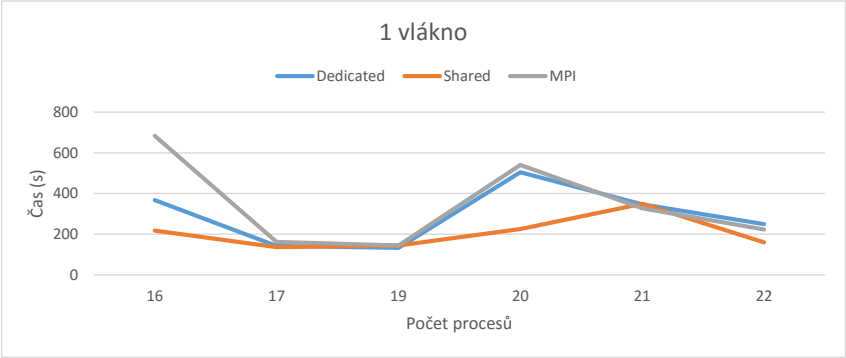


Obrázek 4.13: Škálování vláken při použití scénáře MPI: 16 — 22 procesů

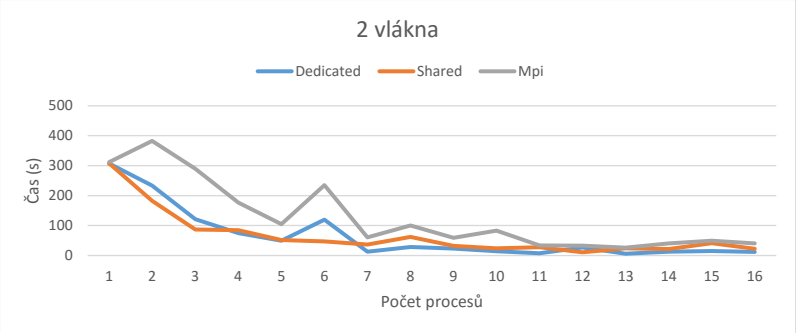
4.2.2 Porovnání dle počtu vláken



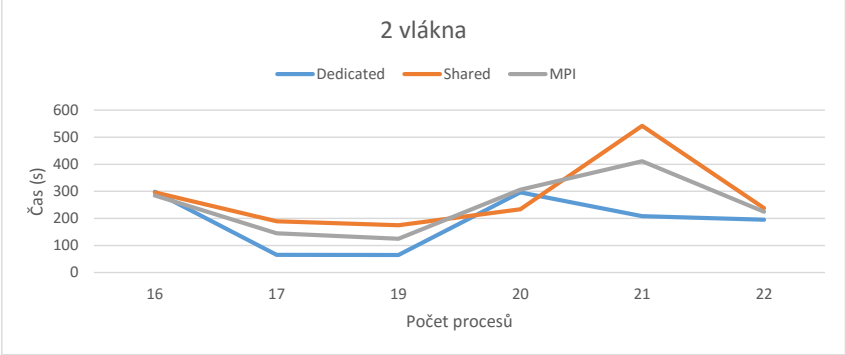
Obrázek 4.14: Porovnání scénářů při 1 vlákně: 1 — 16 procesů



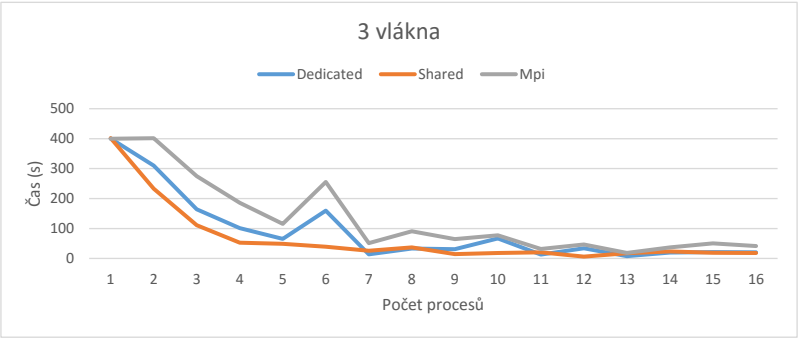
Obrázek 4.15: Porovnání scénářů při 1 vlákně: 16 — 22 procesů



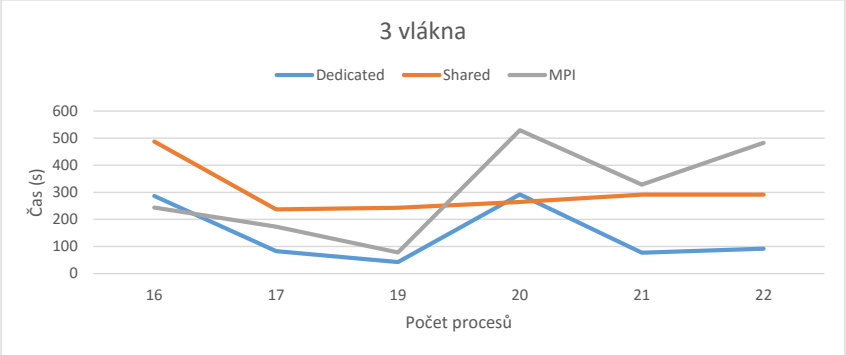
Obrázek 4.16: Porovnání scénářů při 2 vláknech: 1 — 16 procesů



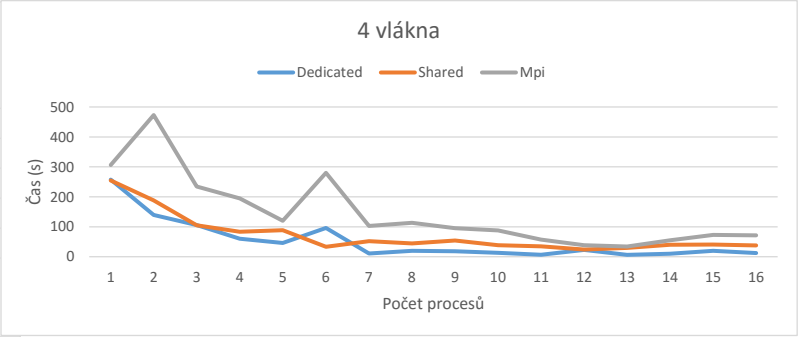
Obrázek 4.17: Porovnání scénářů při 2 vláknech: 16 — 22 procesů



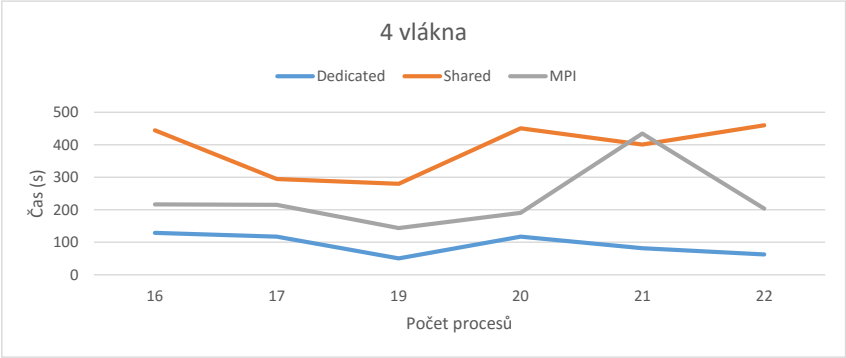
Obrázek 4.18: Porovnání scénářů při 3 vláknech: 1 — 16 procesů



Obrázek 4.19: Porovnání scénářů při 3 vláknech: 16 — 22 procesů



Obrázek 4.20: Porovnání scénářů při 4 vláknech: 1 — 16 procesů



Obrázek 4.21: Porovnání scénářů při 4 vláknech: 16 — 22 procesů

4.3 Vyhodnocení

Test latence ukázal, že moje implementace nezvládá příliš dobře velký počet malých zpráv zasílaných od jednoho procesu ke druhému. Upřímně řečeno jsem překvapen, jak tento test zvládla knihovna Open MPI, u které navíc nebyl výrazně patrný problém se vzrůstajícím počtem vláken. Osobně přikládám tento výsledek využití vyrovnávacích pamětí a seskupování více zpráv do jednoho paketu. U scénáře *Dedicated* se ukázalo, že ačkoliv neškáluje tak dobře jako MPI v rámci procesů, velmi dobře škáluje v počtu vláken zapojených do komunikace. Scénář *Shared* v tomto ohledu zcela propadl, neboť neškáluje ani v rámci procesů, ani v rámci vláken.

U testu škálovatelnost se ukázalo, že scénář *Shared* je nepatrně lepší než scénář *Dedicated* při malém počtu procesů. Dále můžeme pozorovat, že ačkoliv je zprvu implementace MPI pomalejší, s přibývajícím počtem procesů srovnává krok a od počtu zhruba *8* procesů je skoro stejně rychlá jako nové komunikační rozhraní. Všechny implementace vykazují při vysokém počtu procesů nevyváženost, která je ovšem u scénáře *Dedicated* nejnižší. V případě většího počtu procesů pak můžeme ještě pozorovat, že scénář *Dedicated* se nejlépe vypořádává s nárůstem vláken.

U tohoto testu se projevuje jedna zajímavá věc – některé vzrůstající počet procesů nejen, že nemusí urychlit výpočet, ale může ho dokonce výrazně zpomalit. Tento jev nastává z důvodu chybné hašovací funkce, jejíž použití vykazuje výrazné shlukování práce na podmnožině procesů, pokud je počet procesů dělitelný *6*. V případě použití *6* procesů je patrný velký časový skok a menší časový skok je možné pozorovat i v případě počtu *12*. Při použití *18* procesů dokonce test v časovém termínu nedoběhl. Naopak použití vyššího prvočíselného počtu procesů se jeví jako dobrá volba s ohledem na časy *7*, *17* a *19* procesů.

Kapitola 5

Závěr

Cíl práce, tedy návrh a implementace nového komunikačního rohraní s následným porovnání vůči stávajícímu řešení, jsem splnil. Měření neprobíhala přímo s použitím nástroje DIVINE, jak jsem očekával, ale jako nezávislé simulace. Toto selhání je jednak dílem nedostatku času z mé strany, jednak tím, že v průběhu vytváření diplomové práce současně vznikala nová verze nástroje DIVINE, která nebyla připravena na spuštění.

Výsledky měření ukazují, že mnou navržená a implementovaná komunikační vrstva je vhodná pro použití v nástroji DIVINE. Její rozhraní, které na rozdíl od standardu MPI podporuje paralelní přístup, bude doufám jednoduché na použití začlenění do nástroje DIVINE.

Pro budoucí zkoumání této oblasti je vhodné se zaměřit na vytvoření vyrovnávacích pamětí, což by mělo přinést další zrychlení v případě krátkých zpráv. Dalším směrem, kterým by se mohlo zkoumání ubírat, je implementace protokolu nad UDP a její porovnání s mojí implementací a s implementací MPI.

Příloha A

Struktura archivu a demonstrační program

V souboru `archive.zip` se nachází zdrojové soubory nově implementovaného komunikačního rozhraní. Spolu s nimi tam jsou skripty, které slouží k vygenerování testů a jejich spuštění. Pro překlad je potřeba překladač s plnou podporou standardu C++11 a pro generování a spouštění testů je zapotřebí interpret jazyka Python 3. Po nakonfigurování projektu programem `cmake` je možné projekt přeložit.

Ve složce `measurement/execute` se nachází dva skripty. První z nich, `generate.py`, slouží k vygenerování sady testů. Akceptuje jeden až několik parametrů při spuštění, což jsou názvy jednotlivých testů. Pokud je uveden pouze parametr `all`, generátor vygeneruje všechny sady testů. Skript `run.py` slouží ke spouštění testů. Akceptuje jeden až několik parametrů při spuštění, což jsou názvy sad testů, které bude postupně spouštět a ukládat do složek časy běhů.

Demonstrační program přijímá při spuštění několik parametrů. Většina parametrů se dělí na ty, které využívá serverová část, a na ty, které využívá klientská část programu. Jediným společným parametrem je:

`-p` *<port-number>*
: Nastaví číslo portu, na kterém se bude komunikovat. Pokud se neuvede, použije se hodnota *41813*.

A.1 Serverové přepínače

`d, daemon`
: Spustí program v režimu serveru. Pokud není uveden, program se spouští v režimu klienta.

`-l` *<log-file>*
: Nastaví zapisování událostí do logů. Pokud není uvedený, logování

neprobíhá.

`--no-detach`
Spustí server, ale bez démonizace.

A.2 Klientská část

`-h <host-file>`
Soubor se seznamem strojů, které mají být zapojeny do výpočtu. Na každém řádku souboru se musí vyskytovat právě jedna adresa stroje v síti. Povinný parametr.

A.2.1 Příkazy

Program akceptuje příkaz na kterékoliv pozici v přepínačích. Platí ten, který je uvedený jako poslední.

`s, start`
Příkaz, kterým klient spustí serverové instance sama sebe na požadovaných vzdálených strojích. K přístupu využívá *ssh* přístup a funguje pouze na OS GNU/Linux.

`t, status`
Příkaz, pomocí něhož klient zjistí, v jakém stavu se nechází který stroj uvedený v seznamu strojů.

`q, shutdown`
Klient zašle všem strojům ze seznamu příkaz `shutdown`.

`k, forceShutdown`
Klient zašle všem strojům ze seznamu příkaz `ForceShutdown`.

`rs, restart`
Ekvivalent volání s příkazem `k` a `s`.

`r, reset`
Klient zašle všem strojům ze seznamu příkaz `forceReset`.

`c,client`
Program se má chovat jako klient a začít výpočet. Není třeba uvádět, pokud nen9 uveden jiný příkaz.

A.2.2 Výpočty

`table`
Ladící výpis – tabulka otevřených socketů mezi procesy

`load`
: Spustí test škálování ve variantě *Shared*.

`ping`
: Spustí test latence ve variantě *Shared*.

`shared`
: Explicitně specifikuje variantu *Shared*.

`dedicated`
: Explicitně specifikuje variantu *Dedicated*.

`long`
: Nastavuje délku zprávy na *1* KB.

A.2.3 Přepínače

`-n <N>`
: Nastavuje počet vláken testovacího výpočtu. Ovlivňuje také počet otevřených datových spojení mezi procesy.

`-w <N>`
: Nastavuje množství práce.

`-s <N>`
: U testu latencí nastavuje způsob rozhodování o příjemci dotazu. U testu škálování nastavuje zpoždění při generování stavu.

Literatura

[1] Advanced micro devices, inc. x86-64 technology white paper. http://www.amd.com/Documents/x86-64_wp.pdf.

[2] Internet protocol, 9 1981. RFC 791.

[3] Transmission control protocol, 9 1981. RFC 793.

[4] File transfer protocol (ftp), 10 1985. RFC 959.

[5] Intel 80386 programmers's reference manual. http://microsym.com/editor/assets/386intel.pdf, 1986.

[6] A formal description technique based on the temporal ordering of observational behaviour, 1989.

[7] Procedure call standard for the arm architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf, 2015.

[8] P. Kocher A. Freier, P. Karlton. The secure sockets layer (ssl) protocol version 3.0, 8 2011. RFC 6101.

[9] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[10] Jiří Barnat. *Distributed memory LTL model checking*. PhD thesis, PhD thesis, Faculty of Informatics, Masaryk University Brno, 2004.

[11] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[12] R. Branden. Requirements for internet hosts – communication layers, 1 1989. RFC 1122.

[13] J Richard Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[14] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet group management protocol, version 3, 10 2002. RFC 3376.

[15] Franco Callari. The concept of process. http://www.cim.mcgill.ca/~franco/OpSys-304-427/lecture-notes/node4.html, 1996. [Online; navštíveno 7. 01. 2016].

[16] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey C Fox. Mpj: Mpi-like message passing for java. 2000.

[17] Wei-Chen Chen, George Ostrouchov, Drew Schmidt, Pragneshkumar Patel, and Hao Yu. pbdMPI: Programming with big data – interface to MPI, 2012. R Package, URL http://cran.r-project.org/package=pbdMPI.

[18] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[19] EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer Berlin Heidelberg, 2008.

[20] Microsoft Corporation. Processes and threads (windows). https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841%28v=vs.85%29.aspx, 2016. [Online; navštíveno 7. 01. 2016].

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[22] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[23] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[24] S. Deering and R. Hinden. Ip version 6 addressing architecture, 2 2006. RFC 4291.

[25] S. E. Deering. Host extensions for ip multicasting, 7 1986. RFC 988.

[26] R. Fieding and J. Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing, 6 2014. RFC 7230.

[27] MPI Forum. Message passing interface (mpi) forum home page. http://mpi-forum.org/, 2015. [Online; navštíveno 7. 01. 2016].

[28] Al Geist. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[29] Paul C Gilmore. A proof method for quantification theory: its justification and realization. *IBM Journal of research and development*, 4(1):28–35, 1960.

[30] Douglas Gregor and Andrew Lumsdaine. Design and implementation of a high-performance mpi for c# and the common language infrastructure. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 133–142. ACM, 2008.

[31] W Daniel Hillis. *The connection machine*. MIT press, 1989.

[32] IEEE. *Std 1003.1c-1995 Standard for Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1995.

[33] IEEE. *Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE Computer Society Press, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[34] Paul A Karger and Andrew J Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Security and Privacy, 1984 IEEE Symposium on*, pages 2–2. IEEE, 1984.

[35] D Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching, 1968.

[36] Christopher Kohlhoff. Boost. asio. *Online: http://www.boost.org/doc/libs/1*, 48(0):2003–2016, 2003.

[37] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[38] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[39] Chris Lattner. The llvm compiler infrastructure project.

[40] Jochen Liedtke. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995.

[41] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[42] P. Mockapetris. Domain names - concepts and facilities, 11 1987. RFC 1034.

[43] P. Mockapetris. Domain names - implementation and specification, 11 1987. RFC 1035.

[44] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[45] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[46] J. Postel. User datagram protocol, 8 1980. RFC 768.

[47] The Linux Information Project. Process definition by the linux information project (linfo). http://www.linfo.org/process.html, 2006. [Online; navštíveno 7. 01. 2016].

[48] J. Reynolds. Assigned numbers: Rfc 1700 is replaced by an on-line database, 1 2002. RFC 3232.

[49] J. Reynolds and J. Postel. Assigned numbers, 10 1994. RFC 1700.

[50] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[51] R. Hinden S. Deering. Internet protocol, version 6 (ipv6) specification, 12 1998. RFC 2460.

[52] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2013.

[53] C. Lonvick T. Ylonen. The secure shell (ssh) authentication protocol, 1 2006. RFC 4252.

[54] S. Thomson and C. Huitema. Dns extensions to support ip version 6, 12 1995. RFC 1886.

[55] M. Allman V. Paxson and M. Sargent J. Chu. Computing tcp's retransmission timer, 6 2011. RFC 6298.

[56] Vladimír Štill. State space compression for the divine model checker [online], 2013.

[57] Jiří Weiser. Dynamicky rostoucí sdílená hašovací tabulka pro divine [online], 2013.

[58] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.