VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Paralelní a distribuované programování v .NET

# Parallel and Distributed Programming in .NET

2014 Jakub Dolba

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání bakalářské práce

Student: **Jakub Dolba**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Paralelní a distribuované programování v .NET
Parallel and Distributed Programming in .NET

Zásady pro vypracování:

Na katedře je vyvíjen nástroj Kaira. Tento nástroj je určen pro modelování paralelních – distribuovaných aplikací pomocí barevných Petriho sítí. Z těchto modelů pak Kaira umožňuje generovat samostatné aplikace v jazyce C++, které využívají vláken a MPI. Hlavním cílem práce je připravit technologie a nástroje pro budoucí rozšíření Kairy o schopnost generovat cílový kod pro platformu .NET. Zejména atraktivní se jeví možnost využít nějaké existující cloudové řešení. Cíle bakalářské práce lze shrnout v těchto bodech.
1.Seznamte se s různými způsoby vytváření paralelních či distribuovaných aplikací. Zejména se zaměřte na MPI.
2. Kaira je postavena volně dostupných technologiích. Prozkoumejte zejména volně dostupné nástroje a knihovny pro paralelní a distribuované programování, které jsou použitelné na platformě .NET. Zaměřte se zejména na dostupná cloudová řešení.
3. Zvolte nejvhodnější řešení a prakticky realizujte dostatečně rozsáhlý příklad, demonstrující možnosti zvolené technologie.
4. Zhodnoťte přínos zvoleného řešení pro nástroj Kaira.

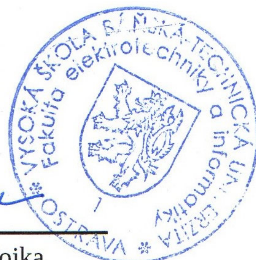
Seznam doporučené odborné literatury:

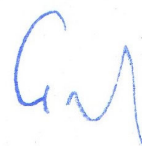
Stanislav Böhm and Marek Běhálek. 2012. Usage of Petri nets for high performance computing. In Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing (FHPC '12). ACM, New York, NY, USA, 37-48.

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

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

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 30. dubna 2014 ..............................

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

V Ostravě 30. dubna 2014 ..............................

Děkuji svému vedoucímu Ing. Marku Běhálkovi, Ph.D. a celému týmu lidí, kteří pracují na nástroji Kaira. Dále bych chtěl poděkovat všem přátelům, kteří mě podpořili a zejména těm, bez kterých bych práci nikdy nedokončil.

**Abstrakt**

Bakalářská práce se zabývá různými způsoby vytváření paralelních či distribuovaných aplikací se zaměřením na MPI. Dále popisuje vyvíjený nástroj Kaira, který je určen pro modelování paralelních - distribuovaných aplikací. Druhá část práce se zaměřuje na možnosti Cloud computingu, zejména pak na možné rozšíření nástroje Kaira o schopnost generování kódu pro cloudová řešení na platformě .NET.

**Klíčová slova:** .NET, Cloud, Cloud computing, Distribuované programování, HPC, Kaira, Message Passing Interface, MPI.NET, Paralelní programování, Petriho sítě, Synchronizace, Windows Azure

**Abstract**

This bachelor thesis deals with different ways of creating parallel or distributed applications focusing on MPI. This thesis also describes developed tool named Kaira. Kaira is tool for modeling of parallel or distributed applications. Second part of thesis is focused on Cloud computing, especially of possible extension the tool Kaira for ability to generate code for cloud solutions based on .NET platform.

**Keywords:** .NET, Cloud, Cloud computing, Distributed programing, HPC, Kaira, Message Passing Interface, MPI.NET, Parallel programming, Petri nets, Synchronization, Windows Azure

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| .NET | – | „dotnet“, soubor softwarových produktů - platforma |
| C# | – | C Sharp, programovací jazyk společnosti Microsoft |
| Cluster | – | počítačový cluster, seskupení výpočetních uzlů(počítačů) |
| GPL | – | General Public Licence, všeobecná veřejná licence |
| HPC | – | High Performance Computing |
| HTTP | – | Hypertext Transfer Protocol, internetový protokol |
| IIS | – | Internet Information Services (Server), webový server společnosti Microsoft. |
| MPI | – | Message Passing Interface |
| OS | – | Operační systém |
| PHP | – | Skriptovací programovací jazyk |
| POSIX | – | Portable Operating System Interface (for Unix) |
| RAM | – | Random Access Memory, paměť s rychlým přístupem |
| SDK | – | Software Development Kit |
| SOA | – | Service Oriented Architecture |
| SOAP | – | Simple Object Access Protocol, síťový protokol |
| SQL | – | Structured Query Language, dotazovací jazyk |
| VB, VB.NET | – | Visual Basic, programovací jazyk společnosti Microsoft |

# Obsah

**1 Úvod** **3**

**2 Vytváření paralelních aplikací** **4**
2.1 Problémy paralelního programování . . . . . 4
2.2 Paralelismus v běžném domácím počítači . . . . . 4
2.3 Organizace paměti v paralelních prostředích . . . . . 5
2.4 Vytváření distribuovaných aplikací . . . . . 6

**3 MPI: Message Passing Interface** **7**
3.1 Komunikace MPI . . . . . 7
3.2 Typy MPI operací . . . . . 7
3.3 Implementace MPI . . . . . 8
3.4 MPI.NET . . . . . 9

**4 Úvod do Petriho sítí** **10**
4.1 P/T (Place / Transition) Petriho sítě . . . . . 10
4.2 Barevné Petriho sítě . . . . . 12

**5 Kaira** **13**
5.1 O nástroji Kaira . . . . . 13
5.2 Integrace sekvenčního kódu . . . . . 14
5.3 Grafické modely a základní chování . . . . . 15
5.4 Shrnutí . . . . . 17

**6 Kaira a Cloud** **19**
6.1 Co je to Cloud . . . . . 19
6.2 Windows Azure . . . . . 21
6.3 Přenos kódu z Kairy do Visual Studia . . . . . 22
6.4 Vystavení a spuštění nativní aplikace na Windows Azure s MPI . . . . . 24
6.5 Zhodnocení . . . . . 29

**7 Závěr** **32**

**8 Reference** **33**

**Přílohy** **33**

**A Obsah přiloženého CD a návod k použití** **34**
A.1 MPI_NET . . . . . 34
A.2 Varianty_Paralelismu . . . . . 34
A.3 WA_HPC_Kaira_Sample . . . . . 34

# Seznam obrázků

1 Příklad průchodu P/T Petriho sítě. . . . . . . . . . . . . . . . . . . . . . . . . 11
2 Vložení kódu do přechodu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3 Příklad: Rozdělení práce mezi ostatní procesy . . . . . . . . . . . . . . . . . . . 16
4 Simulace modelu z Obrázku 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
5 Průnik různých technologií vedoucí k příchodu cloud computingu [5] . . 20
6 Vytvoření aplikace nebo knihovny v Kaiře, převzato se svolením autora z [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

# 1 Úvod

Cílem této bakalářské práce je prozkoumat možnosti cloudových řešení pro budoucí rozšíření vyvíjeného nástroje Kaira, který slouží k modelování paralelních aplikací.

Seznámit se s různými způsoby vytváření paralelní či distribuovaných aplikací je nutným předpokladem pro alespoň základní pochopení problematiky modelování paralelních aplikací. Teorie paralelního programování je značně rozsáhlým odvětvím. Teoretické základy popisuji v 2 kapitole. Poukazuji hlavně na myšlenku paralelního programování a na základní problémy, se kterými je nutné se vypořádat při psaní paralelních aplikací.

Ve 3 kapitole přibližuji standard a specifikaci MPI. Zaobírám se základními prvky standartu a popisuji smysl celého standardu a základních funkcí. V závěru kapitoly se věnuji konkrétní implementaci standardu pro .NET framework.

Po stručném úvodu do Petriho sítí následuje v další kapitole představení samotného nástroj Kaira. Zde popisuji hlavní myšlenku autorů a ukazuji základní funkce a chování programu na konkrétních příkladech.

V posledních kapitolách práce se věnuji především cloudovému řešení společnosti Microsoft a na příkladu demonstruji, jakým způsobem může být Windows Azure využit v dalším vývoji nástroje Kaira.

# 2 Vytváření paralelních aplikací

Paralelizace výpočetních problémů je v dnešní době již běžnou záležitostí. Díky paralelizaci máme možnost snížit čas výpočtů nebo můžeme spouštět paměťově náročné aplikace. Tyto nesporné výhody jsou ovšem zaplaceny složitějším a náročnějším vývojem aplikací v porovnání s jejich sekvenčními variantami. Paralelní počítače se liší v mnoha směrech, mohou to být obyčejné stolní počítače s několika jádry v procesoru nebo také obrovské výpočetní clustry s tisíci procesory.

## 2.1 Problémy paralelního programování

Ať už programujeme paralelní aplikaci běžící na jedno, dvou či více jádrovém procesoru nebo programujeme aplikaci pro výpočetní *cluster*, dříve či později narazíme na některý z problémů, které s sebou přináší psaní paralelních aplikací.

Většinu těchto potíží můžeme označit jako platformně nezávislé, jelikož se obvykle neprojevují jako chyba programu či programovacího jazyka jako takového, ale jedná se o chyby způsobené špatnou logikou či nesprávnou implementací daného algoritmu nebo problému.

Většina těchto problémů souvisí se **synchronizací** mezi jednotlivými vlákny či procesy. Chyby v synchronizaci mohou mít za následek **deadlock** či **nekonzistenci dat**. Úskalí synchronizace se týkají jak aplikací s pár vlákny, tak i složitých paralelních modelů pro vědecké výpočty na superpočítačích s velkým množstvím samostatných procesů.

## 2.2 Paralelismus v běžném domácím počítači

I když v dnešní době procesory s více jádry používají i mobilní telefony (smartphony), *pseudo-paralelismus* na jedno-jádrových procesorech je používán již desítky let. Již na jedno-jádrových procesorech běžely tzv. *multitaskingové* operační systémy, které jsou schopné běhu více aplikací a procesů zároveň. Běh více procesů najednou, tedy *multitasking* na takových procesorech, je umožněn díky plánovači - *scheduler*, který procesům přiřazuje čas na procesoru. Efekt paralelismus či multitasking na jedno-jádrových procesorech je tedy produktem rychlého přiřazování výpočetního času na procesu pro jednotlivé procesy. „Sekvenční procesy" tedy sdílejí jeden procesor a jejich postupné vyhodnocování je dovoleno díky sofistikovaným algoritmům plánování, frontám, nastavování priorit procesů, nastavování stavů procesů a další. To vše má na starost právě operační systém: nejen, že se stará o efektivní využití procesoru, stará se také o paměť využívanou procesy.

Jednotlivé procesy mohou navíc využívat „podprocesů" či „lightwight procesů", které označujeme jako **Vlákna** - *Thread*. Vlákna sdílí zdroje celého procesu, pracují se stejným adresním prostorem a mohou tedy mezi sebou jednoduše komunikovat.

Podstatným rozdílem mezi vláknem a procesem je sdílení paměti; procesy mají přidělenou paměť oddělenou od ostatních procesů, procesy mezi sebou nemohou přímo komunikovat. Vlákna naproti tomu sdílejí všechnu paměť přidělenou procesu.

### 2.2.1 Příklad

Na CD, přiloženém k této práci, je jednoduchý projekt demonstrující řešení stejného problému při využití sériového a paralelního přístupu, podle přílohy A.2 stačí rozbalit soubory a spustit projekt pomocí Microsoft Visual Studio.

Jednoduché konzolová aplikace má za úkol vypočítat všechna prvočísla v daném rozsahu od 1 do 4194304. Využívá jednoduchou implementaci algoritmu pro výpočet prvočísel, Eratosthenovo síto.

Nejprve jsou prvočísla vypočítána hrubou silou v sériovém provedení. Poté jsou prvočísla vypočítána „objektovou" variantou stejného algoritmu, tato varianta je potom také použita pro paralelní variantu, využívající 8 vláken.

Rozdíly mezi jednotlivými řešeními jsou vyjádřeny délkou trvání jednotlivých výpočtů. Výsledky, časy, jednotlivých řešení nejsou příliš zajímavé. Jedná se o neoptimalizované řešení poměrně náročného problému (výpočet prvočísel), nicméně je důležité, jak markantní jsou časové rozdíly jednotlivých řešení. Paralelní řešení může být 3× až 4× rychlejší než sériové/sekvenční řešení, v závislosti na konfiguraci stroje.

## 2.3 Organizace paměti v paralelních prostředích

Paralelní aplikace se nerozdělují pouze podle způsobu napsaného kódu a jeho následném vyhodnocení v procesorech či vláknech. Důležitý je i způsob práce s pamětí a její organizace.[14]

### 2.3.1 Sdílená paměť

Sdílená paměť je přístupná neomezeně všem procesorům. Obvykle jsou to počítače složené z několika procesorů se společnou sběrnicí a společnou pamětí. Problémem je, že s rostoucím počtem procesorů dochází k zahlcení sběrnice. Jako model jsou počítače se sdílenou pamětí vhodné pro malý počet procesorů.

### 2.3.2 Distribuovaná paměť

Každý procesor v systému s distribuovanou pamětí má vlastní soukromou paměť. Výpočetní úlohy tedy probíhají nad lokálními daty a pokud jsou vyžadována data z jiného procesu, resp. paměti, je třeba komunikace mezi procesory. K takové komunikaci je třeba využít systém pro zasílání zpráv mezi procesy (MPI - Kapitola 3 na straně 7). Komunikace v systému s distribuovanou pamětí je obvykle kompletně skryta a jelikož ke komunikaci samotné pak dochází prostřednictvím sítě, má to za následek určité zpoždění.

## 2.4 Vytváření distribuovaných aplikací

Standardem pro výpočty se sdílenou paměti je *OpenMP*. Je zaměřen na paralelizaci smyček. Může být velmi efektivní, pokud lze smyčku vyjádřit jako více nezávislých smyček. Problém nastává ve chvíli, kdy výpočet požaduje sofistikovanější paralelní chování.

Na poli distribuovaných systémů je standardem Message Passing Interface (MPI)[3]. Stručně řečeno, MPI můžeme považovat za rozhraní pro zasílání zpráv mezi jednotlivými výpočetními uzly.

Nicméně v praxi je programování paralelních či distribuovaných aplikací mnohem komplikovanější, než jen odeslání zprávy mezi dvěma výpočetními uzly. Superpočítače jsou často strukturovány jako vzájemně propojené výpočetní clustry se sdílenou pamětí. Využití plného výkonu takového počítače nás vede k hybridním paralelním modelům programování. Nejvyužívanější kombinací je právě spojení OpenMP a MPI.

Problémem takových modelů je ale silná závislost na cílové architektuře. Vyžaduje často rozsáhlé úpravy k dosažení optimálního výkonu na různých superpočítačích. Tento problém se nejčastěji vyskytuje u vědeckých výpočtů, kde není cílem práce vytvořit aplikaci jako takovou, ale získání relevantních výsledků z běhu samotné aplikace. Pokud zvážíme všechny záludnosti paralelního programování, jako např. komplexní návrh a složité ladění aplikace, může být sekvenční řešení problému mnohem efektivnější a výhodnější.

# 3 MPI: Message Passing Interface

Message Passing Interface (MPI) je široce akceptovaný model a standart.[3] MPI je určeno primárně pro komunikaci paralelních programových modelů, ve kterých jsou data odesílána z adresového prostoru jednoho procesu na proces jiný, prostřednictvím společných operací pro každý proces. MPI je *specifikace*, ne implementace. Existuje více implementací MPI, ale MPI není ani programovací jazyk. Všechny operace MPI jsou vyjádřeny jako funkce, rutiny nebo metody. Jsou ale sémanticky nezávislé na programovacích jazycích. Nicméně je MPI vázáno na jazyky C, C++ a Fortran. Standart je definován otevřeně komunitou vědců, vývojových pracovníků a dodavatelů paralelních zařízení.

Hlavním cílem MPI je, jednoduše řečeno, vývoj široce uznávaného standartu pro psaní programů využívající předávání zpráv (message-passing). MPI jako rozhraní by mělo poskytnou praktický, přenosný, výkonný a flexibilní standart pro předávání zpráv.

## 3.1 Komunikace MPI

Odesílání (`SEND`) a příjímání (`RECV`) zpráv jsou dva základní kameny MPI. Téměř každá další MPI funkce může být implementována voláním těchto dvou metod.[3]

Přijímání a odesílání zpráv v MPI funguje následujícím způsobem: Proces *A* se rozhodne odeslat zprávu procesu *B*. Proces A tedy vezme všechna potřebná data a předá je do bufferu pro proces B. Tento buffer bývá popisován jako *obálka zprávy*. Je to stejné, jako když jdeme odeslat dopis na poštu - nejprve jej dáme do obálky, kterou pak zalepíme a teprve potom předáme dopis poště. Jakmile je obálka řádně předána, je za její doručení na správnou adresu plně zodpovědné komunikační zařízení (v případě MPI je to většinou síť). Adresa zprávy je definována identifikátorem procesu - *rank*.

I když je zpráva doručena procesu B, proces B musí dovolit přijetí těchto dat. Jakmile přijetí dat schválí, jsou data přenesena, proces A je obeznámen s dokončením přenosu a může se vrátit zpět k práci.

V některých případech může být odesláno více typů zpráv z jednoho procesu na druhý. Aby se nemusel příjemce probírat všemi zprávami, aby zjistil, co ve zprávě je, MPI dovoluje odesílateli specifikovat identifikátorem typ zprávy - *tag*. Potom může proces přijmout pouze zprávu s určitým tagem, zprávy s jiným tagem budou čekat v bufferu, dokud je proces B nebude schopen zpracovat.

## 3.2 Typy MPI operací

Základním kamenem MPI je komunikace mezi procesy. Základní komunikace vždy probíhá mezi dvěma procesy - *point-to-point* komunikace [3].

Typy point-to-point operací jsou následující:

- blokující
- neblokující

Použití blokující nebo neblokující funkce může mít zásadní vliv na funkci a výkon celé aplikace.

### 3.2.1 Blokující operace

Jak jsme si stručně popsali výše v kapitole 3.1, při odesílání zprávy je obsah zprávy uložen do bufferu - *obálky*. MPI specifikuje metody `SEND` a `RECV` jako blokující, což znamená, že v případě metody `MPI_SEND` nepovolí programu pokračovat, dokud není *obálka* skutečně připravena a odeslána. V případě metody `MPI_RECV` program nepokračuje, dokud není zpráva skutečně přijata a bezpečně uložena.

### 3.2.2 Neblokující operace

S použitím neblokujících operací program nečeká na dokončení přenosu dat a pokračuje v běhu programu. Neblokující operace k odeslání a přijetí zprávy se odlišují prefixem *I* (od slova *immediate* - bezprostředně). `MPI_ISEND` a `MPI_IRECV`.

Výhodou neblokujících operací je, že výpočet může pokračovat bezprostředně po zavolání neblokující funkce bez čekání na dokončení volání, což zvyšuje výkon programu. Neblokující operace dovolují, aby výpočet a komunikace probíhaly současně.

Programátor pak musí ošetřit, zda odeslání/přijetí dat z bufferu bylo dokončeno předtím, než začne s bufferem zase pracovat.

## 3.3 Implementace MPI

Jak jsem již zmínil, MPI je standart a specifikace, nezahrnuje tedy konkrétní implementaci. Samozřejmě ale existují veřejné domény, které volně nabízejí své implementace. MPI je implementováno jako knihovna pro několik programovacích jazyků.

Mezi nejznámější volně šiřitelné implementace MPI patří Open MPI, MPICH (MPICH2), LAM/MPI. Další implementace bývají již různě spojené a upravené verze implementací MPICH a LAM/MPI, obvykle se jedná o komerční implementace společností jako je Intel, Microsoft a další. [3] [9]

Tyto implementace skutečně splňují cíle standartu MPI a jsou opravdu poměrně jednoduše přenosné napříč platformami a jsou značně rozšířené. Těchto implementací vyu-

žívá i Kaira. My se ale budeme dále zabývat implementacemi MS-MPI. Na implementaci MPI.NET pro platformu .NET si demonstrujeme použití MPI.

## 3.4 MPI.NET

MPI.NET je vysoce výkonná implementace MPI pro Microsoft .NET framework. Oproti ostatním implementacím MPI, které poskytují podporu hlavně pro C/C++ a Fortran, MPI.NET poskytuje podporu pro **všechny** jazyky framworku .NET, obzvláště pak pro C#. Zahrnuje navíc několik značných rozšíření, které usnadňují vývoj paralelních programů pro výpočetní clustry.[12]

MPI.NET je postaven na MS-MPI, což je implementace MPI společnosti Microsoft pro *Microsoft HPC Server* a *Windows Compute Cluster Server*. MS-MPI je postaveno na implementaci MPICH2 a umožňuje použít existující kód napsaný pro MPICH2 [13].

### 3.4.1 Instalace MPI.NET

Pro použití MPI.NET nepotřebujeme Windows cluster nebo víceprocesorový počítač. Nicméně bez dalších nástrojů se neobejdeme. (Předpokládá se vývoj na operačním systému Windows XP a vyšší).

Mimo samotného MPI.NET potřebujeme *Microsoft Visual Studio* nebo jiný nástroj schopný kompilace kódů psaných některým z .NET jazyků. Dále je nutné mít nainstalovaný *Microsoft Compute Cluster Pack* nebo *Microsoft Compute Cluster Server*, jak bylo zmíněno MPI.NET je postavený na MS-MPI a bez této knihovny se neobejde.

Instalační balík MPI.NET lze najít na stránce `http://osl.iu.edu/research/mpi.net/software/` [12]. Úspěšnou instalaci lze zkontrolovat spuštěním jednoho z příkladů, které jsou dodány s nainstalovaným MPI.NET balíkem.

Aplikace využívají MPI.NET pak spouštíme, stejně jako běžné MPI aplikace, pomocí příkazu **mpiexec**

```
mpiexec -n <N> <aplikace>
```

Výpis 1: Spuštění MPI.NET aplikace

**<N>** udává počet uzlů, které mají být vytvořeny pro výpočet. **<Aplikace>** je název nebo cesta k aplikaci (Např. HellWorld.exe)

# 4 Úvod do Petriho sítí

Petriho sítě, představují grafický a matematický nástroj pro modelování diskrétních, paralelních či distribuovaných systémů. Při návrhu modelu paralelní aplikace jsou v Kaiře Petriho sítě používány jako vizuální programovací jazyk. Základy Petriho sítí byly položeny již v 60. letech a od té doby byly postupně rozvíjeny. Během vývoje vzniklo několik typů Petriho sítí, představíme si ale pouze základní varianty, které používá nástroj Kaira.5

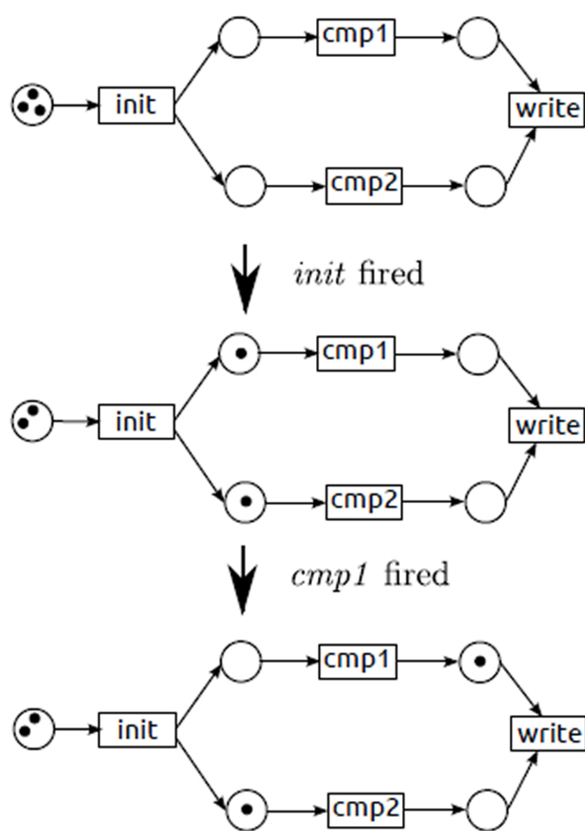
## 4.1 P/T (Place / Transition) Petriho sítě

Skládají se z *míst*, *přechodů* a *orientovaných hran*. Místa jsou znázorňovány jako kroužek, přechody jsou představovány jako obdélníky. Orientované hrany směřují buď od místa k přechodu nebo od přechodu k místu, proto je dále rozdělujeme na *vstupní hrany* a *výstupní hrany*. Hrany nikdy nemohou být přímo mezi dvěma místy nebo mezi dvěma přechody. Hrany by měly mít mít udanou **váhu**, která uvádí, kolik tokenů je třeba po hraně přesunout k provedení přechodu.

Na Obrázku 1 jsou načrtnuty dva stupně vyhodnocování P/T Petriho sítě. **Místa** můžeme považovat za paměťový prostor a **přechody** můžeme považovat za nějakou událost či akci. **Stav** sítě popisujeme jako počet **tokenů** v každém místě. **Token** je obvykle značen jako černá tečka v *místě*. Průchod sítě je proces, během kterého odebíráme a přidáváme tokeny z/do míst. Změny tokenů jsou realizovány v přechodech. Při provádění přechodu je odebrán a zpracován token z vstupního místa přechodu a na výstupní místo je vyprodukován výsledný token. Jeden token je zpracován pro každou vstupní hranu a jeden token je vytvořen pro každou výstupní hranu. Přechod může být proveden pouze tehdy, pokud je na vstupním místě dostatečný počet tokenů; takový přechod se pak nazývá *proveditelný přechod*. **Počáteční stav** sítě udává počet tokenů v jednotlivých místech sítě.

Obrázek 1 modeluje jednoduchý příklad výpočtu rozděleného na dvě části(*cmp1*, *cmp2*), přičemž po dokončení obou části je zobrazen výsledek. Tento jednoduchý výpočet se může dostat do několika různých stavů, v závislosti na pořadí provádění přechodů, avšak počet všech stavů lze popsat počtem tokenů v místech.

Proveditelný přechod může být proveden nezávisle na ostatních přechodech. Pokud síť bude obsahovat pouze **jediný přechod**, pořád můžeme definovat paralelní chování za předpokladu, že na vstupním místě je dostatek tokenů pro vícenásobné provedení přechodu. Tento paralelismus vychází přirozeně z modelu a při navrhování jej nemusíme explicitně definovat.




Obrázek 1: Příklad průchodu P/T Petriho sítě.

## 4.2 Barevné Petriho sítě

V praxi nabízejí P/T Petriho sítě příliš nízko-úrovňové možnosti řešení problému a výsledné modely jsou často velmi rozsáhlé. Rozšířením P/T Petriho sítí můžeme získat prostředky k řešení na vyšší úrovni a jedním z možných rozšíření jsou **Barevné Petriho sítě**.

Hlavní myšlenkou Barevných Petriho sítí je nepovažovat token za „anonymní černou tečku“. V Barevných Petriho sítích můžeme tokenům přiřazovat různé hodnoty. Každé místo má přiřazený svůj typ a tokeny v místech nesou hodnoty podle typu daného místa. V Barvených Petriho sítích tedy místa neobsahují pouze počet tokenů, ale popisují je jako pole hodnot napříč oborem datových typů v jednotlivých místech. Pokud tedy jednotlivé tokeny nesou různé hodnoty, můžeme definovat složitější podmínky, kdy jsou přechody proveditelné a kdy má být přechod proveden. Aby byl přechod proveditelný, musí mít vstupní tokeny „stejnou barvu“.

# 5 Kaira

V této kapitole si představíme vyvíjený nástroj Kaira a možnosti, kterými disponuje.

## 5.1 O nástroji Kaira

Kaira je *open-source* projekt skupiny Verif, jenž je součástí Centra Aplikované Kybernetiky(CAK). Spadá pod GPL licenci. Cílem projektu je vytvořit prostředí vhodné pro vytváření paralelních aplikací se zaměřením na výpočetní clustry(*High-performance computing*).[11]

Poslední verzi Kairy, aktuality a instrukce k instalaci lze najít na webové prezentaci projektu: `http://verif.cs.vsb.cz/kaira/`.[11]

Kaira je samostatné vývojové prostředí nabízející uživatelům potřebnou funkčnost pro vývoj a dokončení aplikací. Umožňuje navrhovat grafické modely paralelních aplikací, vkládat sekvenční kód do uzlů modelu, generovat konečnou aplikaci, zobrazovat simulace či ladit samotný kód.

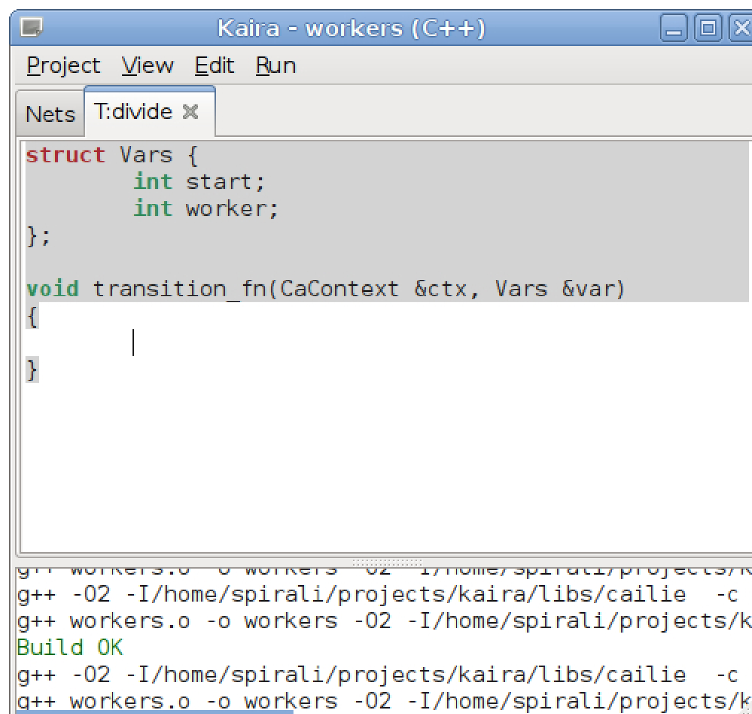
Hlavní myšlenkou projektu je zjednodušit vývoj paralelních aplikací. Kaira je proto vyvíjena jako nástroj s abstraktním modelováním s přirozenou vizualizací a umožňuje jednoduše zakomponovat již existující zdrojový kód. S nástroji jako MPI je často časově náročné vytvořit fungující program, výhodou Kairy je i možnost vytvořit již v prvních fázích vývoje fungující aplikaci, což může umožnit optimalizaci různých knihoven a hardwaru.

Grafické modely paralelních aplikací vytvářené v Kaiře jsou založeny na *Barevných Petriho sítích*. Petriho sítě jsou přijaty jako vizuální programovací jazyk a díky jejich vyjadřovací schopnosti jsou klíčem k vizuálnímu programování paralelních aplikací. [1]

I když Kaira využívá vizuálního programování, není jejím cílem vytvořit celý program pouze pomocí Barvených Petriho sítí. Hlavním účelem vizuálního jazyka je zachytit paralelismus a komunikační aspekty. Sekvenční části programu mohou vytvořeny „klasickým“ jazykem a následně integrovány do vizuálního modelu. Tato vlastnost je pohodlnější nejen pro programátory, ale díky ní lze taky využít již dříve napsaný kód a urychlit tak vývoj prvních prototypů.

Je třeba upozornit, že Kaira není automatický nástroj vyhledávající možný paralelismus v aplikaci. Nicméně umožňuje definovat paralelismus na vyšších úrovních od kterých následně může odvodit implementační detaily.

Kaira umožňuje vkládat sekvenční kód v C++ a vytváří aplikace využívající vlákna a MPI. Právě možnost vytvářet samostatné aplikace odlišuje Kairu od podobných nástrojů, které také disponují možnostmi jako abstraktní modelování, simulace a vizuální




Obrázek 2: Vložení kódu do přechodu

programování. Z tohoto pohledu je tedy cílem vytvořit nástroj podobný jiným grafickým návrhovým prostředím, jako je NetBeans nebo Visual Studio.

Než se podíváme na vlastnosti Kairy samotné, je třeba mít alespoň základní informace o Petriho sítích.

## 5.2 Integrace sekvenčního kódu

Jak je zmíněno v kapitole 5.1, Kaira dovoluje využití již napsaného sekvenčního kódu. Jednou z možností, jak to provést, je vložit kód do *přechodů* či *míst*. Kaira pomáhá programátorovi vygenerováním šablony funkce podle vytvořené struktury modelu. Na Obrázku 2 je ukázán editor Kairy, který nedovolí změnit šablonu funkce, ale lze napsat jakýkoli kód do těla funkce. Šablona ukazuje kód přechodu *devide* z Obrázku 3

Další možností, jak zakomponovat kód do navrhovaného modelu, je integrace typů a funkcí C++. Lze využít datové typy reprezentující matici z existující C++ knihovny jako typ tokenů a funkce knihovny lze použít na hrany. Je nutné pouze definovat funkce pro serializaci takových typů a určit, jak se mají hodnoty zobrazovat během simulace.

## 5.3 Grafické modely a základní chování

V kapitole 5.1 již bylo uvedeno, že grafické modely v Kaiře jsou založeny na Barvených Petriho sítích. Barevné Petriho sítě jsou vhodně rozšířeny a upraveny pro jednodušší modelování algoritmů pro distribuované systémy. Pro popis vytvořených modelů autoři Kairy navrhli vlastní jednoduchý jazyk **NeL** (Net Language), protože je jednodušší pracovat s méně sofistikovaným jazykem, který usnadňuje překlad a generování konečného kódu pro různá koncová zařízení[1]. Jazyk by měl být lehce pochopitelný pro čtenáře se základními zkušenostmi s běžnými programovacími jazyky[1].

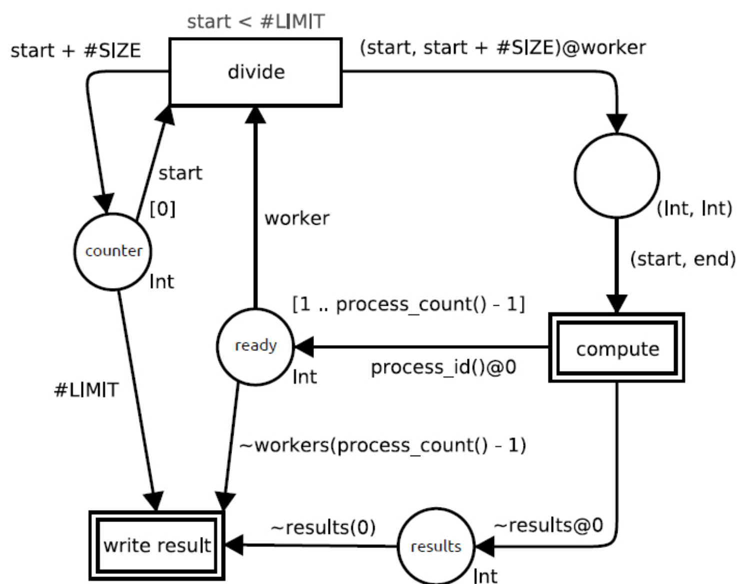
### 5.3.1 Síťové procesy modelů

Jaké jsou tedy rozdíly modelovacího jazyka v porovnání s Barevnými Petriho sítěmi? V sémantice Kairy může existovat více kopií celé sítě. Tyto kopie se nazývají **síťové procesy**. V praxi pak bude jedna kopie sítě v jednom výpočetním uzlu. Během simulace lze samozřejmě počet síťových procesů změnit. Každý síťový proces je zcela samostatný s vlastními stavy a běží nezávisle na ostatních. Vytváření tokenů je jedinou možnosti, jak komunikovat mezi různými síťovými procesy. Přechody ovšem mohou zpracovávat tokeny z lokálního síťového procesu, proto testování, zda je přechod proveditelný, vyžaduje pouze informace z lokálního síťového procesu. Tato sémantika vychází z povahy distribuované paměti, to znamená, že proces může odeslat zprávu, ale nemůže přímo číst z paměti jiného procesu.

Smyslem síťových procesů je popsat stav distribuované paměti, kde každý síťový proces představuje jeden adresový prostor. Každý síťový proces tedy běží paralelně k ostatním. Síťové procesy se vytváří pro každý MPI proces. Paralelní chování síťového procesu není omezováno - pokud jde spustit více vláken v jednom MPI procesu, lze vykonat více úkonů současně bez jakéhokoli zásahu do modelu.

### 5.3.2 Chování modelu

Na Obrázku 3 je demonstrováno chování modelu při běžném problému, kde hlavní proces rozděluje úkoly mezi ostatní procesy. Pokud jeden z procesů dokončí přiřazený úkol, zažádá hlavní proces o nový úkol. Toto se opakuje dokud neskončí celý výpočet. V tomto příkladu jsou jednotlivé úkoly intervaly čísel. Typy tokenů v *místech* jsou zobrazeny dole napravo, *stav* při inicializaci je zobrazen napravo nahoře (tvar přechodů a míst, v kapitole 4 na straně 10). Podmínky proveditelnosti přechodu jsou zobrazeny nad přechodem. Přechod je proveden pouze pokud je podmínka splněna. Hrany jsou navíc rozlišovány jako *normální* nebo *balené* hrany. Normální hrana zpracuje/vytvoří vždy jeden token. Balená



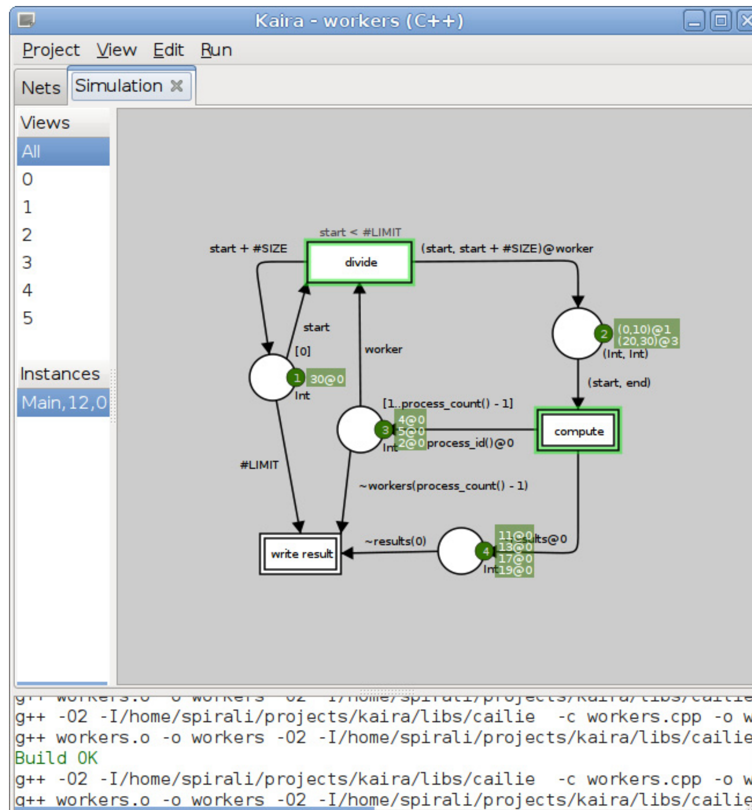

Obrázek 3: Příklad: Rozdělení práce mezi ostatní procesy

hrana může pracovat s více tokeny najednou a je od normálních hran rozlišena prefixem ~ (tilda). Vstupní balená hrana zpracuje všechny tokeny z místa a hodnoty tokenů jsou předávány v poli. Výstupní balená hrana z pole vytváří vždy jeden token pro každou hodnotu z pole.

Výše jsem zmiňoval, že při spouštění modelu existuje více běžících kopií(síťové procesy). Ty jsou identifikovány jedinečným celým číslem začínajícím od nuly a výše. Funkce `process_id` vrací identifikační číslo současného síťového procesu. Počet všech procesů lze získat funkcí `process_count`. V příkladu na obrázku 3 místo *ready* představuje čekající **workery** (pracovníky) a místo *counter* udržuje informaci o začátku následujícího intervalu. Přechod *devide* rozděluje intervaly mezi workery tak, že odebere čekajícího workera a odešle jej s intervalem na jeho síťový proces. Výrazy `start` a `worker` jsou proměnné, jejich hodnota je přiřazena po provedení přechodu.

Výstupní hrany mohou specifikovat cílový síťový proces výrazem za znakem `@`. Pokud není tento výraz nastaven, je token vytvořen ve stejném procesu ve kterém byl proveden. Výrazy `#LIMIT` a `#SIZE` jsou konstanty, které jsou nastaveny při startu vyhodnocování celého modelu.

Přechod *compute* představuje výpočet intervalu samotného. Dvojitý okraj znamená, že uvnitř přechodu je definována nějaká C++ funkce. Jakmile je výpočet přechodu dokončen, pak je výsledek poslán zpět na síťový proces 0 a zároveň je token s identifikátorem síťového procesu vložen na místo *ready*. Pokud hodnota v místě *counter* dosáhne




Obrázek 4: Simulace modelu z Obrázku 3

nastavené mezní hodnotě `#LIMIT` a všechny procesy dokončí jejich výpočet, pak jsou všechno tokeny z místa *results* odebrány a vypsány. Přechod *write results* obsahuje funkci, která výsledky vhodně vypíše.

## 5.4 Shrnutí

Představili jsme si tedy nástroj Kaira. Přiblížili jsme si vizuální jazyk založený na Barevných Petriho sítích, které vhodně rozšiřuje a následně využívá pro modelování paralelních aplikací pro Výpočetní clustry. [1]

Nástroj je stále aktivně vyvíjen. Autoři stále experimentují i se základní myšlenkou celého projektu a stále hledají nejvhodnější cestu, jak vhodně řešit složitější paralelní problémy. V budoucnu by mohla Kaira sloužit nejen jako koncept, ale skutečně usnadnit vývoj paralelních aplikací, zjednodušit práci s paralelními algoritmy či zprostředkovat vhodné prostředí pro programátory bez hlubších znalostí paralelních technologií.

Zatím je Kaira vyvíjena pro Unixové prostředí, z experimentální povahy projektu není ihned nutné vyvíjet či optimalizovat nástroj pro jiné prostředí (Windows, atd.), jelikož je neustále vyvíjeno i samotné jádro. Přesto má Kaira potenciál k dalšímu rozvoji a rozšíření funkcionality i pro jiné platformy a programovací jazyky.

# 6 Kaira a Cloud

Projekty vytvořené v nástroji Kaira lze spouštět a ladit na běžném domácím počítači. Hlavní myšlenkou Kairy je ale možnost spustit náročné úlohy na více výpočetních uzlech, superpočítači či výpočetních farmách.

Ne každý uživatel nástroje bude mít přístup k takovému výpočetnímu centru. V dnešní době je proto velmi lákavé a jednoduché využít některé z cloudových řešení. Cloudy jsou schopny poskytnout přinejmenším stejný výkon jako dostupné výpočetní farmy, na které můžeme narazit například na vysokých školách, přičemž finanční náklady zůstanou v přijatelných mezích.

V této kapitole bych chtěl stručně přiblížit pojem Cloud a na cloudovém řešení společnosti Microsoft demonstrovat možnosti, které nabízí pro nástroj Kaira.

## 6.1 Co je to Cloud

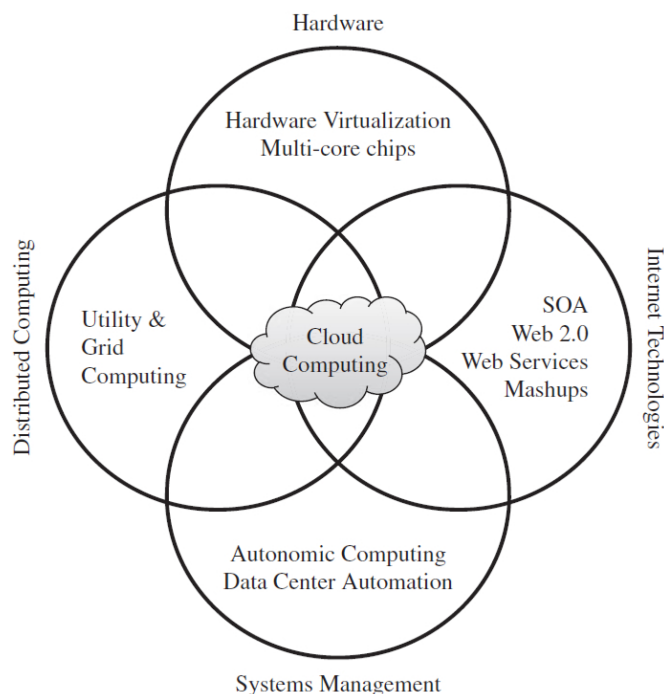
Pojem Cloud nebo Cloud computing zastřešuje kategorii výpočetních nástrojů dodávajících určitý výkon na požádání. Označuje výpočetní infrastrukturu, která je vnímána jako „cloud“, z něhož mají uživatelé přístup k aplikacím z libovolného místa na světě. Hlavním principem tohoto modelu je nabídnout výpočetní sílu, úložiště a software jako službu.

Na Obrázku 5 je znázorněno spojení jednotlivých odvětví, jejichž vývojem jsme se dostali až ke Cloud computingu. Teprve značný zájem velkých firem zapříčinil rozvoj jednotlivých odvětví, postupné standardizování dovolilo průnik odvětví, což vedlo k širšímu spojení v cloud computing. Z mnoha technologií informačního průmyslu lze vybrat několik zastřešujících oborů, které měly přímý vliv na vývoj cloudu, jako jsou distribuované výpočty s využíváním výpočetních farem, Internetové technologie (Web 2.0, webové služby), rostoucí požadavky na hardware, virtualizace a správa systémů.

### 6.1.1 Dostupné cloudy

Dnes již existuje nepřeberné množství cloudových služeb a řešení. Nicméně ne všechny jsou stejně flexibilní a mohou se zaměřovat pouze na konkrétní problém, například poskytování úložiště pro uživatele nebo jen doručování obsahu po celém světě.

Cloudových řešení, která by poskytovala prostředí pro spouštění vlastních aplikací a zároveň umožňovala využít svou architekturu pro *High Performance Computing* není zase tak mnoho. Z dlouhého seznamu cloudů jsem vybral pouze ty, které jsou dle mého názoru vhodné jako „náhradní řešení“ místo fyzických výpočetních center.




Obrázek 5: Průnik různých technologií vedoucí k příchodu cloud computingu [5]

- Windows Azure
- Amazon EC2 (Elastic Compute Cloud)
- Google Compute Engine (Google App Engine)
- Heroku
- RackSpace

Každý z uvedených cloudových poskytovatelů má své přednosti. V současné době lze dohledat mnoho nezávislých Benchmark testů, porovnávajících skutečný výkon a náklady mezi jednotlivými poskytovateli. Poměr ceny k výkonu v této oblasti ale nemusí ihned znamenat dosažení lepších výsledků. Zaměřil jsem se na Windows Azure, protože mimo jiné nabízí celou svou infrastrukturu a platformu jako službu a s vývojovým prostředím Visual Studio umožňuje pohodlné testování a vystavení aplikace přímo na Azure.

## 6.2 Windows Azure

Azure je platforma a operační systém realizující ideu Cloud computingu v podání společnosti Microsoft. Konceptem platformy je nabídnou vývojářům flexibilitu cloudu pro různorodé požadavky na architekturu vyvíjených služeb.

Od roku 2008, kdy byl Azure představen, doznala platforma značných změn a vylepšení. Datová centra se nachází po celém světě a každý rok se otevírají další, aby byla zaručena co nejlepší dostupnost služeb.

Windows Azure nabízí široký rozsah možností využití jeho služeb. Od doručování obsahu (např. vysílání živých přenosu), přes běžné webové servery, datová úložiště, virtualizaci, až po využití infrastruktury pro HPC. Aktuality, portál pro správu, dokumentaci, ukázky a další užitečné informace lze najít na domovské stránce Windows Azure: `http://www.windowsazure.com`

### 6.2.1 Podpora, Interoperabilita

Azure umožňuje, stejně jako další poskytovatelé cloudých služeb, vývoj v mnoha programovacích jazycích, ke kterým průběžně doplňuje a aktualizuje SDK. Nicméně hlavní zbraní celé platformy je samozřejmě plná podpora .NET Frameworku.

Kromě .NET platformy můžeme využít i SDK pro následující jazyky:

- Java
- Python
- PHP
- Ruby
- Node.js

„Hlavním" jazykem je ale samozřejmě jeden z jazyků .NET, tzv. Role Entry Point, což je rozhraní mezi aplikací a Azure prostředím, musí být napsáno v C# nebo VB. [15]

Windows Azure je operační systém a poskytuje nezbytné služby a rozhraní pro hostování aplikací v cloudu. Samotné aplikace proto běží na instancích virtuálních serverů, jako je Windows Server. Nicméně lze vytvořit instance i Linuxových serverů.

Jelikož máme ve výsledku přístup k plnohodnotnému serveru, je teoreticky možné využít jakýkoli programovací jazyk. Proto lze spouštět aplikace napsané čistě v C++, například jako „Joby" pomocí Windows Server HPC Manageru.

### 6.2.2 Virtual Machine a Windows Azure Cloud Service

Všechny výpočetní zdroje ve Windows Azure jsou tedy poháněny virtuálními stroji. Ne každý virtuální stroj je ale stejný.

Azure umožňuje vytvořit a používat plnohodnotný *Virtual Machine* - Virtuální stroj, který je plně v našich rukách. Obraz tohoto stroje zabírá místo v našem úložišti, můžeme jej libovolně upravovat, musíme se sami start o aktualizace a běžnou údržbu. Všechny změny jsou prováděny v našem lokálním úložišti, jakékoli selhání nebo chyba může znamenat nenávratnou ztrátu dat.

Konfigurovat virtuální stroje kvůli výpočetním instancím může být zdlouhavé, zbytečně náročné a drahé. *Windows Azure Cloud Service* nabízí skutečnou „infrastrukturu jako službu". Základní myšlenkou je „pošli mi kód a já ho za tebe spustím". Toho dosahuje prostřednictvím Rolí. Tyto role lze označit za *výpočetní instance* a v současné době jsou používány 2 role:

- Worker Role
- Web Role

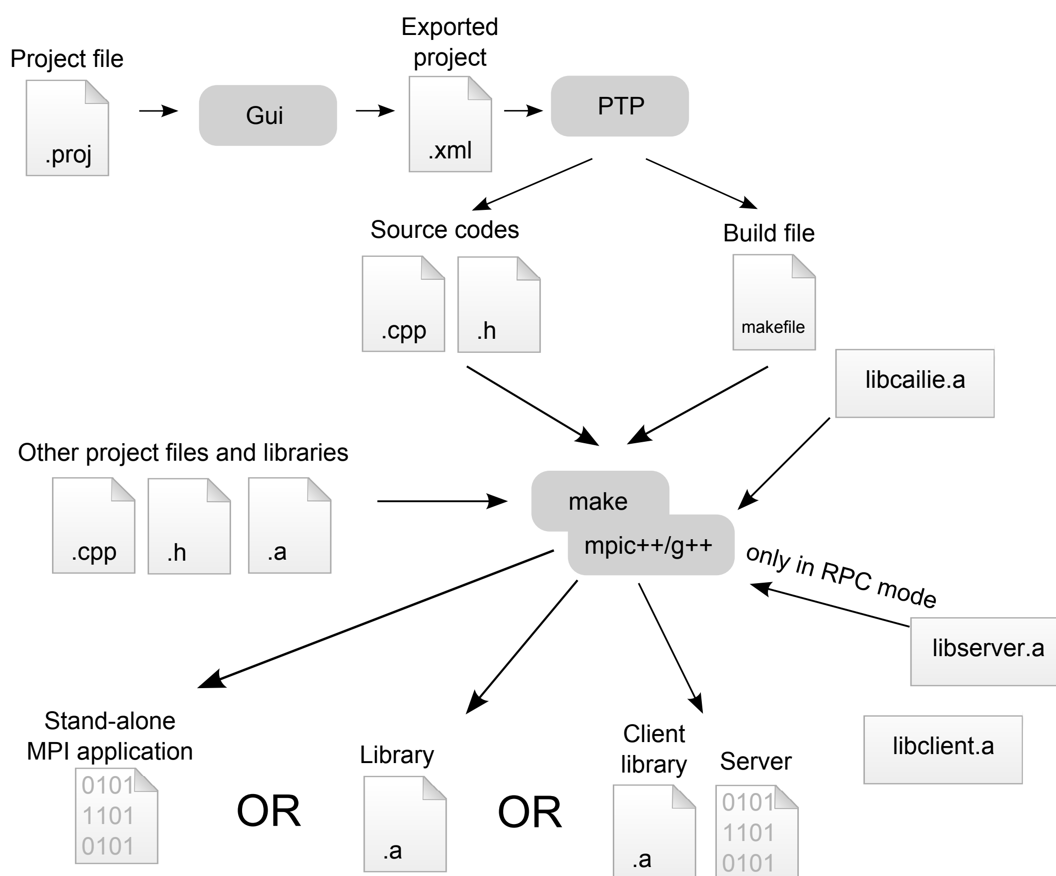
**Worker Role** funguje jako hrubá pracovní síla na pozadí. Je to obraz správně nastaveného systému spravovaného Microsoftem. Je to jednoznačně identifikovatelný hostitelský stroj nezávislý na ostatních rolích, které jsme vytvořili. Lze vytvořit mnoho instancí stejného stroje a ty mohou mezi sebou komunikovat. Je to tedy výpočetní uzel v pomyslné virtuální výpočetní farmě.

**Web Role** funguje stejně jako Worker Role, s tím rozdílem že po spuštění je nakonfigurovaný a spuštěný IIS, tedy webový server - Front End. Worker Role lze tedy označit za Back End.

## 6.3 Přenos kódu z Kairy do Visual Studia

Jednou z hlavních myšlenek této práce je spuštění kódu vygenerovaného nástrojem Kaira v prostředí Windows Azure. Kaira je připravena pro možnost generování cílového kódu projektu v jiných jazycích než je C/C++. Jak je zmíněno v kapitole 5, nástroj je stále vyvíjen a to včetně jádra samotného, proto zatím nemá smysl optimalizovat nástroj pro generování kódu pro jiné platformy.

Kaira ovšem při sestavování projektu generuje *.cpp* a *.h* soubory s čistým zdrojovým kódem, obsahující veškerý kód nutný k běhu aplikace/projektu (viz. obrázek 6). Tyto soubory tedy jsou použity při pokusech přenosu hotového projektu z Kairy na Windows Azure.




Obrázek 6: Vytvoření aplikace nebo knihovny v Kaiře, převzato se svolením autora z [2]

Pro sestavení aplikace je potřeba jádro *Cailie* a Kairou vygenerované zdrojové kódy podle vytvořeného projektu. Zmíněné zdrojové soubory jsem chtěl přímo využít pro přenos a spuštění vygenerované aplikace ve Visual Studiu, následně pak ve Windows Azure. Nejdůležitější částí je samozřejmě přenos kódu jádra. Bohužel se mi nepovedlo zdrojové kódy z Kairy spustit v alespoň částečně fungující podobě. Nepodařilo se mi odstranit problémy se závislostí na Unixovém prostředí a najít odpovídající alternativy použitých knihoven. Další z obtíží při přenosu je minimální dokumentace jádra.

Přestože se mi nepodařilo spustit projekt Kairy na Windows Azure, zaměřil jsem se alespoň na spuštění jednodušší aplikace využívající MPI.

## 6.4 Vystavení a spuštění nativní aplikace na Windows Azure s MPI

Možností, jak spustit nativní aplikaci s MPI, existuje na Windows Azure několik. Nicméně ne každá využije plně potenciál celé platformy. Po několika úspěšných i neúspěšných pokusech jsem se zaměřil na variantu, která podle mě nejlépe vystihuje myšlenku spojení Kairy a cloudového řešení.

V následující části textu si tedy ukážeme jednu z cest, kterou lze použít pro vystavení nativní aplikace na Windows Azure, používající MPI a její následné spuštění na více výpočetních instancích pomocí HPC Manageru.

Na CD, přiloženém k této práci, je projekt využívající předpřipravený formulář a rutiny, které umožňují jednodušší a rychlejší vystavení aplikace přímo z Visual Studia. Zároveň konfiguruje celý projekt tak, aby bylo možné téměř okamžité spuštění aplikace pomoci HPC Manageru. Projekt na CD naleznete podle instrukcí v příloze A.3

Následující postup má za úkol nastínit, jakým způsobem si představuji budoucí rozšíření nástroje Kaira o možnost připravit hotový projekt pro spuštění v cloudu. Zároveň ukazuje závislosti, bez kterých není možné fungující MPI aplikaci na Windows Azure spustit.

### 6.4.1 Příprava řešení

Před otevřením samotného projektu je potřeba připravit a nainstalovat některé artefakty, bez kterých se deploy nepodaří. Pro úspěšné spuštění řešení je potřeba:

1. Windows 7 Profesional (nebo Ultimate), Windows 8
2. Microsoft Visual Studio 2012
3. Windows Azure SDK for .NET

4. Windows Azure Tools for Microsoft Visual Studio 2012
5. Windows Azure HPC Scheduler SDK (v1.8)

```
C:\Program Files\Windows Azure HPC Scheduler SDK
```

6. HPC Pack 2012 R2 MS-MPI + client utilities

```
c:\Program Files\Microsoft HPC Pack 2012
```

7. Předplatné na portálu Windows Azure (alespoň aktivovaný měsíční Free Trial)

Všechny SDK a doplňující balíky lze stáhnout prostřednictvím Microsoft Download Center: http://www.microsoft.com/en-us/download/default.aspx?navIndex=1

### 6.4.2 Konfigurace řešení

Ve složce „WA_HPC_Kaira_Sample“ spustíme prostřednictvím Microsoft Visual Studio 2012 soubor „Azure_Kaira_Sample.sln“

Celé řešení by mělo být připraveno k okamžitému sestavení a následnému spuštění. Přesto je potřeba zkontrolovat několik věcí:

1. Jako **Start-Up** projekt je nastaven projekt **AppConfigure** v podsložce **Deployment Application**
2. Solution configurations jsou nastavené na **Debug**
3. Solution platforms jsou nastavené na **Mixed Platforms**
4. Projekt „Workers_Kaira_Project“ a všechny projekty v podsložkách „SOA Sample“ jsou nastaveny jako nedostupné.

Projekt **AppConfigure** spustí jednoduchý formulář pro konfiguraci služeb a konečné vystavení služeb.

### 6.4.3 Spuštění řešení a konfigurace služby

Po spuštění řešení se spustí formulář **Configure and Publish a Sample Windows Azure Application**, který vyplníme. Popis formuláře je v tabulce 1.

Upozorňuji, že **Service name** se jménem a heslem administrátorského účtu je nutné si dobře zapamatovat.

| Položka | Popis |
|---|---|
| **Subscription ID** | ID předplatného Windows Azure |
| **Management certificate** | Certifikát nainstalovaný v lokálním počítači a zároveň nahraný v portálu Windows Azure (Settings, Managment Certificates). Formulář umožňuje certifikát vygenerovat. |
| **Service name** | Jedinečný DNS název služby. |
| **Location** | Primární geografické umístění služeb Windows Azure. |
| **Administrator account information** | Jméno a heslo potřebné pro pozdější přihlášení ke spuštěným službám (vzdálená správa serveru). |
| **Number of nodes** | Počet jednotlivých uzlů. Změníme pouze počet **Compute** uzlů 4 |

Tabulka 1: Popis formuláře pro deploy aplikace

Po vyplnění správných údajů již stačí potvrdit pomocí **Configure** a poté vystavit aplikaci na Windows Azure pomocí tlačítka **Publish**.

Konfigurace i nahrání dat na Azure bude chvíli trvat a k samotnému spuštění služeb dojde také až po pár minutách. Průběh vytváření jednotlivých instancí a služeb lze sledovat na webovém portálu Windows Azure.

#### 6.4.4 Spuštění MPI aplikace na Windows Azure

Než spustíme příklad nativní MPI aplikace **bakery.exe**, musíme ověřit zda byla aplikace správně nahraná.

##### 6.4.4.1 Kontrola nahrané aplikace

1. Otevřeme Windows Azure portál. V seznamu všech služeb musí být pod vámi uvedeným **Service name** spuštěny služby: Cloud service, Storage Account a SQL Database.

2. Otevřeme Cloud service a v záložce **Instances** označíme jeden z Compute nodů. Připojíme se na uzel pomocí tlačítka **Connect**, prostřednictvím správy vzdálené plochy.

3. Po připojení k vzdálenému serveru zkontrolujeme, zda jsou nahrány následující soubory ve složce:

```
E:\approot\Bakery.exe
E:\approot\Bakery.ilk
E:\approot\Bakery.pdb
```

4. Dále zkontrolujeme stejným způsobem připojení k **Head node**. Zde zkontrolujeme, zda je v uzlu nahrán i HPC plugin. V Head node musí existovat následující složka:

```
E:\plugins\HpcHeadNode\HPCPack
```

#### 6.4.4.2 Konfigurace firewallu

Po kontrole uzlů se připojíme prostřednictvím vzdálené správy na **Head node** a v příkazové řádce vytvoříme výjimku pro naši aplikaci bakery.exe:

```
clusrun /nodegroup:computenode hpcfwutil register bakery.exe e:\approot\bakery.exe
```

Výstupem tohoto příkazu bude potvrzení o přijetí výjimek na všech uzlech. Například pro 4 výpočetní uzly potvrzení vypadá zhruba takto:

```
---------------------------- COMPUTENODE1 returns 0
    ----------------------------
Successfully registered application bakery.exe
---------------------------- COMPUTENODE3 returns 0
    ----------------------------
Successfully registered application bakery.exe
---------------------------- COMPUTENODE4 returns 0
    ----------------------------
Successfully registered application bakery.exe
---------------------------- COMPUTENODE2 returns 0
    ----------------------------
Successfully registered application bakery.exe

---------------------------- Summary
    ----------------------------
4 Nodes succeeded
0 Nodes failed
```

### 6.4.5 Spuštění MPI aplikace

Po kontrole vystavení a konfiguraci firewallu můžeme pomocí správy vzdálené plochy na **Head node** spustit aplikaci bakery.exe

1. Připojíme se pomocí správy vzdálené plochy z portálu Windows Azure na **Head node**

2. Otevřeme příkazovou řádku

3. Vytvoříme *job* pomocí příkazu:

   ```
   job submit /numnodes:4 mpiexec e:\approot\bakery.exe 100000 1000
   ```

   Poznámka: spouštíme aplikaci na 4 uzlech, další parametry aplikace bakery.exe udávají intenzitu a počet výpočtů pro vyčíslení číslo *pí*. Pokud budeme mít vytvořený jiný počet výpočetních uzlů, musíme specifikovat *job /numnodes* správným počtem uzlů.

   Výstupem příkazu může být následující:

   ```
   Job has been submitted. ID: 3.
   ```

4. Stav můžeme sledovat pomocí příkazu:

   ```
   job view <jobID>
   ```

5. Výstup z aplikace můžeme zobrazit pomocí příkazu:

   ```
   task view <jobID>
   ```

   Pro aplikaci bakery.exe vypadá výstup takto:

   ```
   Task Id                    : 3.1
   State                      : Finished
   Task Name                  :
   Command Line               : mpiexec e:\approot\bakery.exe 100000 1000
   Resource Request           : 4-4 nodes
   Allocated Nodes            :
   COMPUTENODE1,COMPUTENODE2,COMPUTENODE3,COMPUTENODE4
   Exit Code                  : 0
   Error Message              :
   Output                     :

   Windows Azure HPC Scheduler          Time:12:43:49

   Congratulations!! You calculated pi 1000 times in 2.752520 seconds.
   Processors Used For This Job
   ----------------------------
   Process 0 on COMPUTENODE1
   ```

```
Process 1 on COMPUTENODE2
Process 2 on COMPUTENODE3
Process 3 on COMPUTENODE4

PI is approximately 3.1415926535981167, Error is 0.0000000000083236

Start Time          : 3/16/2014 12:43:45 PM
End Time            : 3/16/2014 12:43:49 PM
Total Kernel Time   : 249
Total User Time     : 1983
```

## 6.5 Zhodnocení

Možnosti jak použít nativní kód ve vystavované aplikaci na Windows Azure je několik. Lze jej využít pro zpracování časově náročnějších úloh na pozadí, zpracování dávkových úloh a podobně. Nicméně použití nativního kódu, nebo i řízeného kódu, zároveň s MPI je záležitost složitější a je nutný zcela jiný přístup k řešení tohoto problému.

### 6.5.1 Azure a MPI

Možnost spouštět na Azure platformě aplikace založené na MPI přišla s uvedením Microsoft HPC Pack 2008. Implementace MS-MPI, kterou využívá, umožňuje volání všech běžných funkcí podle standartu MPI.

V příkladech jsem využíval *HPC Pack 2012 MS-MPI* a vyzkoušel jsem několik základních funkcí MPI, které využívá i Kaira. Některé funkce jsem otestoval přímo na Azure, jiné na příkladech v implementací s MPI.NET (kapitola 3.4), protože i tato ve výsledku volá funkce samotného MS-MPI.

MS-MPI používané v HPC Packu a tedy i na Windows Azure plně podporuje funkce pro kolektivní komunikaci, jako `MPI_Barrier` či `MPI_Reduce`. Důležitá je také podpora *point-to-point* komunikace, na které závisí většina operací v Kaiře. Lze využít blokující i neblokující varianty funkcí, tedy `MPI_Send`, `MPI_ISend` a `MPI_Recv`, `MPI_IRecv`.

Pro samotnou MPI komunikaci aplikace mezi uzly ve Windows Azure je nutné vytvořit výjimky pro firewall, což dovolí jednotlivým uzlům mezi sebou vzájemně komunikovat. Konfiguraci firewallu je nutné provést v příkazovém řádku po připojením se na **Head node**.

Spuštěné MPI aplikace lze trasovat za použití argumentu **-trace** příkazu **mpiexec**. Chybové zprávy aplikace používají k identifikaci uzlu **host name**, což je celý název uzlu a bez použití HPC Cluster Manageru může být složité jej identifikovat .

Jednotlivé MPI joby nelze rozdělit mezi rozdílně nasazené Windows Azure uzly. Nelze například využít dvě rozdílné šablony pro vytvoření výpočetních uzlů a poté je společně využít pro stejnou práci. Rozdílně nasazené uzly jsou izolovány a MPI procesy nejsou schopny, v tomto případě, mezi sebou komunikovat. Tento problém lze částečně eliminovat využitím skupin - node groups.

### 6.5.2 Azure a Kaira

Migrace Kairy z Unixového prostředí na Windows Azure se ukázala jako značně komplikovaná záležitost, nicméně je možné zjednodušit přenos zdrojových kódů několika způsoby.

Při přenosu zdrojových souborů je nutné nahradit některé použité knihovny a nahradit některé Unixové knihovny jejich Win32 alternativami. Například jsem se pokusil použít knihovnu **Pthreads-win32**, která implementuje velkou podmnožinu standardních POSIX funkcí pro práci s vlákny a k nim související rozhraní. Projekt Pthreads je vyvíjen jako Open Source a lze jej nalézt zde `http://www.sourceware.org/pthreads-win32/`.

Po několika pokusech použití dalších externích knihoven jsem zjistil, že tento přístup je zbytečně složitý a vnáší do celého řešení spoustu dalších komplikací, jako například neaktuálnost použitých knihoven či na první pohled ne příliš zřejmé závislosti k dalším externím zdrojům.

#### 6.5.2.1 Přepsání jádra Kairy přímo pro Win32

Nejrozumnější, ale nejnáročnější řešení je přepsání a sestavení samotného jádra *Cailie* v prostředí Windows. V souvislosti s tímto řešením by bylo potřeba lehce upravit Kairou vygenerovaný kód jednotlivých projektů.

Samozřejmě se zcela odstraní závislost na unixovém prostředí a umožní se tak plně využít sílu výkonných HPC Cluster Serverů. Hlavní nevýhodou je, že Kaira je stále ve vývojové fázi a udržení kompatibility dvou jader (pro Unix a Windows) je samo o sobě velmi náročné, nemluvě o přepsání samotného jádra.

#### 6.5.2.2 Subsystem for UNIX-based Appliacions

Nejslibnějším řešením, které se nepodařilo včas realizovat, je podle mě využit **Subsystem for UNIX-based Appliacions** (SUA). SUA poskytuje abstraktní operační systém pro POSIXové procesy. Poskytuje tedy

kompletní UNIXové prostředí pro spuštění unixových aplikací běžící na některém z vybraných operačních systémů Windows. SUA podporuje a přináší case-sensitive jména souborů, kompilační nástroje a více než 300 UNIX příkazů, nástrojů a skriptů. SUA je instalován odděleně on jádra Windows a proto nabízí plnou funkcionalitu UNIXu bez emulace prostřední. [4]

SUA je připraven ke kompilaci a spouštění unixových zdrojových souborů v prostřední rodiny Windows Server systémů. SDK a další utility jsou navíc dostupné pouze v systémech Windows 7 Ultimate a Windows 8 Pro. V nižších variantách tohoto systému nelze SUA spustit.

Nevýhodou tohoto řešení je, že přidáváme další vrstvu mezi naši aplikaci a systém, což může vést k rapidnímu poklesu výkonu celé aplikace.

# 7 Závěr

Seznámení se s způsoby vyvíjení paralelních aplikací je nutným základem této práce. MPI navíc do celého odvětví vývoje paralelních a distribuovaných aplikací vnáší nepřeberné množství možností. Náročnost vývoje těchto aplikací mi pomohla ocenit myšlenku a pochopit základní princip nástroje Kaira. Je patrné, že nástroj má velký potenciál a v dnešní době, kdy je paralelizace a práce s distribuovanými systémy již běžnou záležitostí, najde své uplatnění.

Rostoucí využití cloud computingu má již dnes značný dopad na svět High-Performance computingu. Spojení cloudu a výpočetních clustrů je velmi mocnou kombinací, která nabízí nové možnosti. Možnosti HPC závisí na výpočetní síle, kterou jsme schopni poskytnout a Windows Azure disponuje obrovským množstvím této síly. Využití těchto prostředků je tedy více než vhodné. Platforma Windows Azure nabízí poměrně jednoduchý systém pro širokou škálovatelnost infrastruktury a spolu s podporou několika programovacích jazyků je flexibilita celého systému skvělou příležitostí využít jej místo výpočetních clustrů nebo superpočítačů, které nemusí být vždy jednoduše dostupné.

Hlavním cílem práce bylo připravit technologie pro budoucí rozšíření Kairy pro platformu .NET za využití některého z cloudových řešení. Z dostupných cloudů jsem vybral řešení společnosti Microsoft, Windows Azure. I když se mi nepodařilo upravit vygenerovaný projekt Kairy a následně jej spustit na platformě Windows Azure, demonstruji možnosti platformy na příkladu nativní aplikace využívající MPI. MPI lze označit za základní stavební kámen Kairy. Relativní jednoduchost, rychlost vystavení a následné spuštění aplikace mě utvrdilo v přesvědčení, že využití cloudu jako cílové „architektury“ pro Kairu je krok správným směrem.

Jelikož je v současné době nástroj Kaira stále ve vývoji a Windows Azure nabízí mnoho možností využití a je skutečně flexibilním cloudovým řešením, nedává momentálně příliš smysl zaměřit se na rozšiřování nástroje o možnost generování kódu pro tuto platformu. Jednoznačným negativem je totiž rozdíl mezi prostředím Windows a Unix. Unixové prostředí, v němž je Kaira vyvíjena a je připravena pro toto prostředí generovat cílový kód, je značně odlišné od celého universa prostředí Windows. Tento rozdíl nemusí být nutně nepřekonatelnou překážkou pro možné rozšíření Kairy.

Jakub Dolba

# 8 Reference

[1] Stanislav Böhm and Marek Běhálek *Usage of Petri nets for high performance computing. In Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing (FHPC '12)*, ACM, New York, NY, USA, 37-48, 2012.

[2] Stanislav Böhm *Unifying Framework For Development of Message-Passing Applications*, FEI VŠB-TUO Ostrava, 2013 .

[3] MPI Forum *Dokumentace MPI*, http://www.mpi-forum.org, 2008.

[4] *MSDN - Microsoft Developer Network* [online]. Dostupné z: http://msdn.microsoft.com, 2013.

[5] BUYYA, Rajkumar, James BROBERG a Andrzej GOSCINSKI *Cloud computing: principles and paradigms*, Hoboken, N.J.: Wiley, 2011.

[6] REDKAR, Tejaswi a Fabio Claudio FERRACCHIATI *Windows Azure platform*, New York: Distributed by Springer-Verlag New York, 2009.

[7] Edward G. Coffman Jr., M. J. Elphick, Arie Shoshani *System Deadlocks*, ACM Comput. Surv., 67-78, 1971.

[8] DAVID CHAPPELL *Windows HPC Server and Windows Azure, High-Performance Computing in the Cloud*, DavidChappell & Associates, 2010.

[9] RNDr. Ondřej Jakl, CSc. *Programming of Parallel Applications, Study support for distance learning*, Ostrava, 2003

[10] MARKL, J. *Učební texty k předmětu Petriho sítě I.*[online]. [cit. 2013-04-16]. Dostupné z: http://www.cs.vsb.cz/markl/pn/index.html

[11] *Webová prezentace projektu Kaira* [online]. [cit. 2013-04-20]. Dostupné z: http://verif.cs.vsb.cz/kaira

[12] *MPI.NET: High-Performance C# Library for Message Passing* [online]. [cit. 2013-04-25]. Dostupné z: http://osl.iu.edu/research/mpi.net/

[13] *Microsoft High Performance Computing for Developers* [online]. [cit. 2013-04-28]. Dostupné z: http://msdn.microsoft.com/en-us/library/ff976568.aspx

[14] Ing. Pavel Krömer Ph.D. *Učební texty předmětu Paralelní a distribuované systémy* [online]. [cit. 2013-04-29]. Dostupné z: http://homel.vsb.cz/~kro080/pds/

[15] *Dokumentace Windows Azure, MSDN* [online]. [cit. 2013-02-18]. Dostupné z: http://msdn.microsoft.com/en-us/library/windowsazure/dd163896.aspx

# A Obsah přiloženého CD a návod k použití

Na disku přiloženém k této práci jsou tyto soubory:

**dol0039_bp.pdf** Elektronická verze tohoto dokumentu

**Priloha.zip** Archiv s několika adresáři a připravenými projekty

Po rozbalení archivu nalezneme v 3 složky, každá obsahující jedno řešení spustitelné pomocí Microsoft Visual Studio 2012.

## A.1 MPI_NET

Triviální příklady ukazující použití MPI.NET

## A.2 Varianty_Paralelismu

Jednoduchá konzolová aplikace, porovnávající rychlost stejného algoritmu pro výpočet prvočísel. Demonstruje rozdíl mezi sériovým a paralelním řešením problému.

## A.3 WA_HPC_Kaira_Sample

Řešení sestavené z několika veřejných zdrojů, demonstrující ideu rozšíření nástroje Kaira o možnost generování cílového kódu pro Windows Azure. Popis a spuštění tohoto řešení je popsán v této práci.