Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Testování knihovny fork/join v jazyce Java 7

Bakalářská práce

Karel Maxa

Školitel: RNDr. Jaroslav Icha

České Budějovice 2012

Bibliografické údaje

Maxa Karel, 2012: Testování knihovny Fork/join v jazyce Java 7.

[Fork/join testing in Java 7. Bc. Thesis, in Czech.] – 47 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Abstract:

In release Java 7 comes new library fork/join from package java.util.concurrent, which help to better usage of parralel architecture and thus bring more time effective applications. The bachelor is aimed at describe and make recommendation for usage of this library. Main part of bachelor is aimed at implementation tests and their evalution for showing time efficiency. For the test was written program in Java, which was executed on different system platforms and different hardware architectures.

Abstrakt:

S verzí Java 7 přichází nová knihovna fork/join v balíčku java.util.concurrent, která pomůže lépe využít vícejádrovou architekturu a tím přináší časové zefektivnění běhu aplikací. Práce se zabývá popsáním a zpracováním doporučení pro použití této knihovny. Hlavní část práce se zabývá provedením testů a jejich následném vyhodnocení za účelem ilustrace časové efektivity při použití této knihovny. Pro testy byl napsán program v jazyce Java, který byl spouštěn napříč různými systémovými platformami a hardwarovými architekturami.

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích, dne 12. 4. 2012

Podpis: .....

Poděkování

Na tomto místě bych rád poděkoval především svému školiteli RNDr. Jaroslavu Ichovi za cenné rady a připomínky, pomoc při řešení problémů a odborné vedení.

Moje poděkování také patří Mgr. Fibichovi za rady ohledně výpočetního clusteru, virtuální organizaci MetaCentrum za poskytnutí výpočetních clusterů, rodičům, přítelkyni a všem, kteří mně během studia jakkoliv pomohli.

Slovníček pojmů

Framework – softwarová struktura, která může obsahovat (algoritmy, knihovny…) a tím podporuje vývoj aplikací.

Třída – představuje skupinu objektů, které nesou stejné vlastnosti.

Instance – konkrétní datový objekt odvozený z třídy.

Konstruktor – speciální metoda třídy, která se volá vždy, když je vytvářena instance této třídy.

Abstraktní třída – třída, z které nelze vytvářet instance, ale může být nadtřídou jiných tříd.

Statická třída – třída, která nemá žádné instanční proměnné ani metody a pro využití metod, které obsahuje, není potřeba vytvářet instanci.

Podtřída – třída, která využívá dědičnosti a je tedy podtřídou své nadtřídy. Podtřída obsahuje všechny veřejné proměnné a metody nadtřídy.

Výpočetní cluster – seskupení počítačů, které spolu komunikují po síti a navenek vystupují jako jeden počítač.

Seznam obrázků

1 Úvod

1.1 Úvodem

Předmětem této práce je popsání nového frameworku obsaženého v nejnovější verzi Java 7 a otestování časové efektivity při jeho nasazení na počítačích s vícejádrovými procesory.

Vícejádrové procesory jsou dnes široce rozšířené napříč servery, desktopy a notebooky. Také se začínají objevovat v mnohem menších zařízeních, jako jsou chytré telefony a tablety. Tyto procesory otevírají nové možnosti ve vícevláknovém programování, protože vlákna programu mohou být prováděna na několika jádrech současně. Paralelní běh aplikace dokáže plně využít procesor a tím šetří celkový čas, potřebný pro její vykonání. (1)

Jazyk Java podporuje vlákna a souběžnost od první verze. Java zahrnuje synchronizační primitiva jako je *synchronized*[I] a *volatile*[II] a knihovnu, která obsahuje třídu *Thread.* Nicméně primitiva pro souběžnost, které byla účelné v roce 1995, odrážely realitu hardwaru tehdejší doby. Většina systémů vůbec nepodporovala paralelismus.

S tím jak vícejádrové systémy zlevnily, potřebovalo více aplikací využít paralelní architekturu. Do verze Java 5 byl přidán balík *java.util.concurent*, který poskytuje sadu užitečných komponent pro vytváření aplikací pracujících paralelně: paralelní kolekce, fronty a vlákna. Tyto mechanismy byly vhodné pro programy s rozumnou rozdělitelností úkolu na menší části. Práce aplikací musí vždy být rozdělena tak, aby počet souběžně běžících vláken nebyl nikdy nižší, než je počet jader procesoru.

S ohledem na Moorův zákon nebudou mít procesory v budoucnosti vyšší frekvence ale větší počet jader na čipu. Při vstupu do éry mnoha jader potřebujeme najít algoritmus pro lepší rozdělení práce nebo budeme riskovat, že procesory zůstanou nečinné, když je potřeba udělat spousta práce. Je nutné, aby pokroky v hardwaru zachytily softwarové platformy.

[I] Synchronizační primitivum vztahující se k metodám, které zajišťuje, že metoda bude současně prováděna pouze jedním vláknem.

[II] Synchronizační primitivum vztahující se k proměnným, které indikuje, že hodnota proměnné bude modifikována různými vlákny.

Tento problém řeší Java 7, která obsahuje framework obsahující třídy s paralelními algoritmy, které lépe rozdělují práci mezi vlákna. Tento framework se nazývá fork/join framework. (2)

1.2 Cíle práce

Prvním z hlavních cílů, které si tato práce klade je poskytnout popis knihovny a doporučení, která usnadní zájemcům její použití.

Druhým hlavním cílem je ilustrovat časovou efektivitu nasazení knihovny fork/join pomocí vhodně zvolených algoritmů.

1.3 Metodika

1.3.1 Použitá testovací metodika

Testy byly prováděny na dvou nejrozšířenějších systémových platformách (Windows a Linux) a především na třech počítačích s různým počtem jader procesoru (2, 4 a 8 jader). Dále byly testy provedeny na výpočetním clusteru s platformou Linux, kde byl počet jader roven 16 jádrům, respektive 24 jádrům. Testovací program byl spouštěn z příkazové řádky. Při spouštění testovacího programu nebyl spuštěn žádný jiný program, kromě programu VisualVM (viz. 3.3) a počítač byl odpojen od sítě. Při spouštění programu na výpočetním clusteru (16 a 24 jader) nebyl spuštěn žádný jiný program a cluster byl připojen k internetu. Testovací program obsahoval dva různé algoritmy. Prvním byl algoritmus pro rozmazání obrázku. Druhým byl algoritmus pro výpočet párové energie molekul vody. Pro algoritmus rozmazávající obrázek byly parametry určující velikost rozmazání rovny 20, 50, 100. Algoritmus provádějící výpočet párové energie molekul vody vždy proběhl pro 1000 a 10000 molekul vody. Pro každý parametr bylo provedeno 10 testů, kde naměřené hodnoty byly zaneseny do tabulky, následně byly zprůměrovány a byla vypočtena směrodatná odchylka. Poté byly vypočteny dva parametry zrychlení. Prvním parametrem bylo zrychlení doby výpočtu při využití fork/join frameworku oproti výpočtu, který nevyužíval žádný paralelismus. Druhým parametrem bylo zrychlení doby výpočtu při využití fork/join frameworku oproti výpočtu, který využíval knihovnu *ExecutorService* obsaženou v Java 6. Oba parametry zrychlení byly zaneseny do samostatných grafů. Grafy byly rozděleny podle platforem. Souběžně s testy proběhlo měření vytížení procesoru, využití paměti a počtu

aktivních vláken. Pro tento účel byl použit program VisualVM, kde výstupy z tohoto programu byly vizualizovány graficky.

Měřen byl vždy pouze čas, který byl potřebný pro provedení výpočtu. Čas, který byl potřebný pro načtení obrázku nebo pro načtení molekul ze souboru nebyl zahrnut.

1.3.2 Použitá testovací technika

Notebook Asus A6JE-AP018M – 2 jádra

- Procesor: Intel Core2 Duo T5500 1.66 GHz
- Paměť: 2560 MB DDRII (2 GB + 512 MB)
- Pevný disk: 100 GB PATA, 5400 otáček
- Operační systém 1: Windows 7 SP1 x86 (32 bit)
- Operační systém 2: Ubuntu 11.10 Oneiric Ocelot
- Java: Oracle Java SE 1.7.0_02

Desktop Dell Optiplex 980 - 4 jádra

- Procesor: Intel Core i5 Duo 650 3,2 GHz
- Paměť: 2x 4096 MB DDR3 1333MHz
- Pevný disk: 500 GB SATA, 7200 otáček
- Operační systém 1: Windows 7 SP1 x64 (64 bit)
- Operační systém 2: Ubuntu 11.10 Oneiric Ocelot
- Java: Oracle Java SE 1.7.0_02

Notebook MSI GT640-076XCZ – 8 jader

Procesor má fyzicky 4 jádra. Využívá technologii HyperThreading[III] a tedy má 8 logických jader.

- Procesor: Intel Core i7 720QM 1.6 GHz
- Paměť: 2x 2048 MB DDR3 1066 MHz
- Pevný disk: 500 GB SATA, 7200 otáček
- Operační systém 1: Windows 7 SP1 x64 (64 bit)

[III] Technologie vyvinutá firmou Intel, která pomocí aktivace dvou řídících jednotek v procesoru vytváří z jednoho fyzického procesoru dva logické procesory.

- Operační systém 2: Ubuntu 11.10 Oneiric Ocelot
- Java: Oracle Java SE 1.7.0_02

Výpočetní uzel Tarkil 12.1 a Tarkil 15.1 – 16 jader

V obou výpočetních uzlech byl využit jeden procesor, který obsahuje 4 fyzická jádra. Oba procesory využívají technologii HyperThreading[IV]. Celkem tedy bylo využito 16 logických jader.

Specifikace jednoho uzlu:

- Procesor: 2x Intel Xeon X5570 2.93GHz
- Paměť: 24 GB
- Pevný disk: 2x 300GB SAS, 15000 otáček
- Operační systém: Debian GNU/Linux
- Java: Oracle Java SE 1.7.0_03

Výpočetní uzel Konos 4 a Konos 7 – 24 jader

V obou výpočetních uzlech byl využit jeden procesor, který obsahuje 6 fyzických jader. Oba procesory využívají technologii HyperThreading[IV]. Celkem tedy bylo využito 24 logických jader.

Specifikace jednoho uzlu:

- Procesor: 2x Intel Xeon X5680 3.33GHz
- Paměť: 24 GB
- Pevný disk: 2x 300GB SATA
- Operační systém: Debian GNU/Linux
- Java: Oracle Java SE 1.7.0_03

1.3.3 Použité programové vybavení

Pro vizualizaci využití systémových prostředků byl použit nástroj VisualVM, který je součástí JDK 6 a vyšší. Nástroj VisualVM je napsaný v jazyce Java, proto je možné jej spouštět na platformách, které Javu podporují.

Pro spouštění testovacího programu na systému Windows i Linux byla používána příkazová řádka.

[IV] Technologie vyvinutá firmou Intel, která pomocí aktivace dvou řídících jednotek v procesoru vytváří z jednoho fyzického procesoru dva logické procesory.

2 Fork/join framework

2.1 Úvod

Fork/join paralelismus je jedna z nejjednodušších a nejefektivnějších návrhových technik pro získání dobrého paralelního výkonu. Fork/join algoritmy jsou paralelní verzí známých algoritmů „rozděluj a panuj“, s typickou podobou:

```
Vysledek vyres(Problem problem) {
    if (problem je maly) {
        vyres problem primo
    }
    else {
        rozdel problem na nezavisle casti
        zajisti asynchoronni provedeni vsech casti (fork) a pockej na jejich vyreseni
    (join)
        proved kompozici vysledku z vysledku dilcich casti
    }
}
```

Operace *fork* spouští nové paralelní fork/join podúkoly. Operace *join* způsobí, že aktuální úkol nebude pokračovat, dokud rozdělený dílčí úkol nebude splněn. Fork/join algoritmy, stejně jako ostatní „rozděluj a panuj“ algoritmy, jsou téměř vždy rekurzivní, kdy opakovaně rozdělují dílčí úkoly, dokud nejsou dostatečně malé pro řešení pomocí jednoduchých, krátkých sekvenčních metod. (3)

2.2 Návrh frameworku

Jádrem fork/join frameworku je nový exekutor *ForkJoinPool,* rozšiřující *AbstractExecutorService,* který umožňuje běh instancí, které jsou podtřídou třídy *ForkJoinTask.*

2.2.1 ForkJoinPool

Třída *ForkJoinPool* se liší od ostatních tříd rozšiřujích *AbstractExecutorService* především tím, že využívá *work-stealing* algoritmus. Třída *ForkJoinPool* je konstruována s danou cílovou úrovní paralelizace, která je rovna počtu dostupných jader procesoru. Pro vytvoření instance třídy *ForkJoinPool* je možno využít konstruktor bez parametru, kde velikost fondu vláken bude rovna počtu dostupných procesorů nebo konstruktor s parametrem, kde tuto velikost přesně specifikujeme. Fond (Pool) se snaží udržet dostatečný počet vláken dynamickým přidáváním, pozastavením nebo obnovováním vnitřních pracovních vláken, dokonce když některé úkoly čekají na provedení ostatních.
Třída *ForkJoinPool* obsahuje tři základní metody pro provedení instancí třídy *ForkJoinTask:*

- *execute(ForkJoinTask)* – Zajišťuje (asynchronní) provedení daného úkolu.
- *invoke(ForkJoinTask)* – Provede zadaný úkol a vrátí výsledek po jeho dokončení.
- *submit(ForkJoinTask)* – Zajišťuje (asynchronní) provedení daného úkolu a vrátí objekt typu *Future*[V], z kterého lze získat výsledek.

Kromě metod pro provádění instancí třídy *ForkJoinTask* a metod pro kontrolu jejich životního cyklu, poskytuje třída *ForkJoinPool* metody pro kontrolu stavu, které jsou určené pro pomoc při vývoji, ladění a monitorování fork/join aplikací. (4)

[V] Třída Future představuje v jazyce Java výsledek asynchronního výpočtu.

2.2.2 ForkJoinTask

ForkJoinTask je základní abstraktní třída pro úkoly, které běží v rámci instance třídy *ForkJoinPool*. *ForkJoinTask* je třída podobná třídě *Thread*, ale je mnohem méně paměťově náročná. Velké množství úkolů a dílčích úkolů může být umístěno v malém počtu skutečných vláken v instanci třídy *ForkJoinPool*.

Abstraktní třída *ForkJoinTask* má dvě podtřídy. Těmito podtřídami jsou třídy *RecursiveAction* a *RecursiveTask*. Třída *RecursiveAction* je podtřídou třídy *ForkJoinTask<void>* a je používána v těch případech, kdy prováděný algoritmus v metodě *compute* nevrací žádný výsledek. Třída *RecursiveTask<V>* je podtřídou třídy *ForkJoinTask<V>* a je používána v těch případech, kdy prováděný algoritmus v metodě *compute* vrací výsledek, kde datový typ výsledku je roven datovému typu generického parametru *V*.

Důvodem pro používání *RecursiveAction* nebo *RecursiveTask<V>* místo jejich nadtřídy je, že v konkrétní *ForkJoinTask* podtřídě jsou deklarovány proměnné, které tvoří její parametry. Dále je definována metoda *compute()*, která nějakým způsobem používá kontrolní metody dodané nadtřídou. I když tyto metody mají veřejný modifikátor viditelnosti, některé z nich mohou být volány pouze z jiných *ForkJoinTasks*. Takže pokoušení se je vyvolat v jiných kontextech může mít za následek výjimky pravděpodobně zahrnující *ClassCastException*.

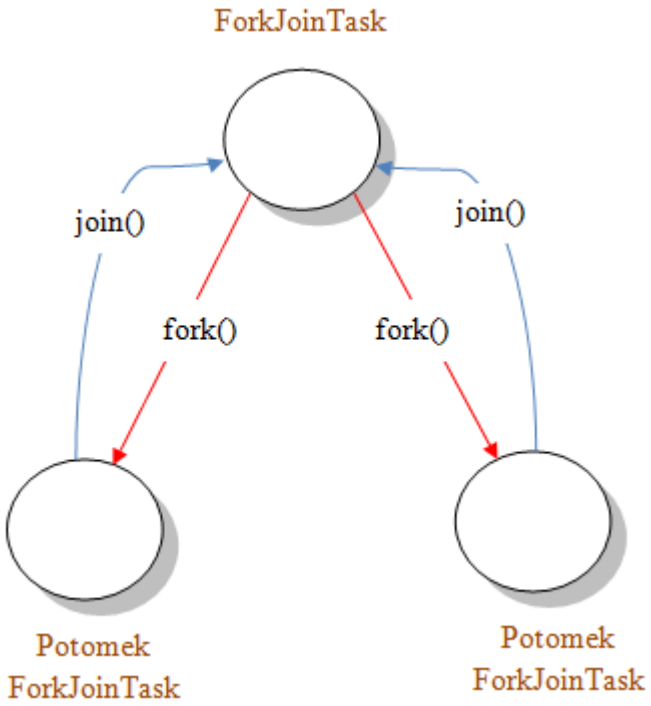
Vylepšená účinnost *ForkJoinTasks* vychází ze sady omezení, které odráží jejich určené použití ve výpočetních úkolech využívajících funkce nebo pracujících s izolovanými objekty. Výpočty by se měly vyhnout *synchronized* metodám, neměly by provádět blokující IO[VI] operace a v ideálním případě by měly mít přístup k proměnným, které nejsou přístupné ostatním úkolům. Částečné porušení těchto omezení může být v praxi přípustné, ale může mít za následek snížený výkon. (5)

Instance, které jsou podtřídou třídy *ForkJoinTask* mají dvě důležité metody:

- Metoda f*ork()* umožňuje objektům tříd, které jsou podtřídou třídy *ForkJoinTask* plánovat asynchronní provedení. To umožňuje, že nová instance může být spuštěna z již existující instance.
- Metoda j*oin()* umožňuje instancím čekat na dokončení práce instancí, které byly spuštěny pomocí metody *fork()*.

[VI] IO (intup-output) je vstupní/výstupní operace (například čtení/zápis do souboru)

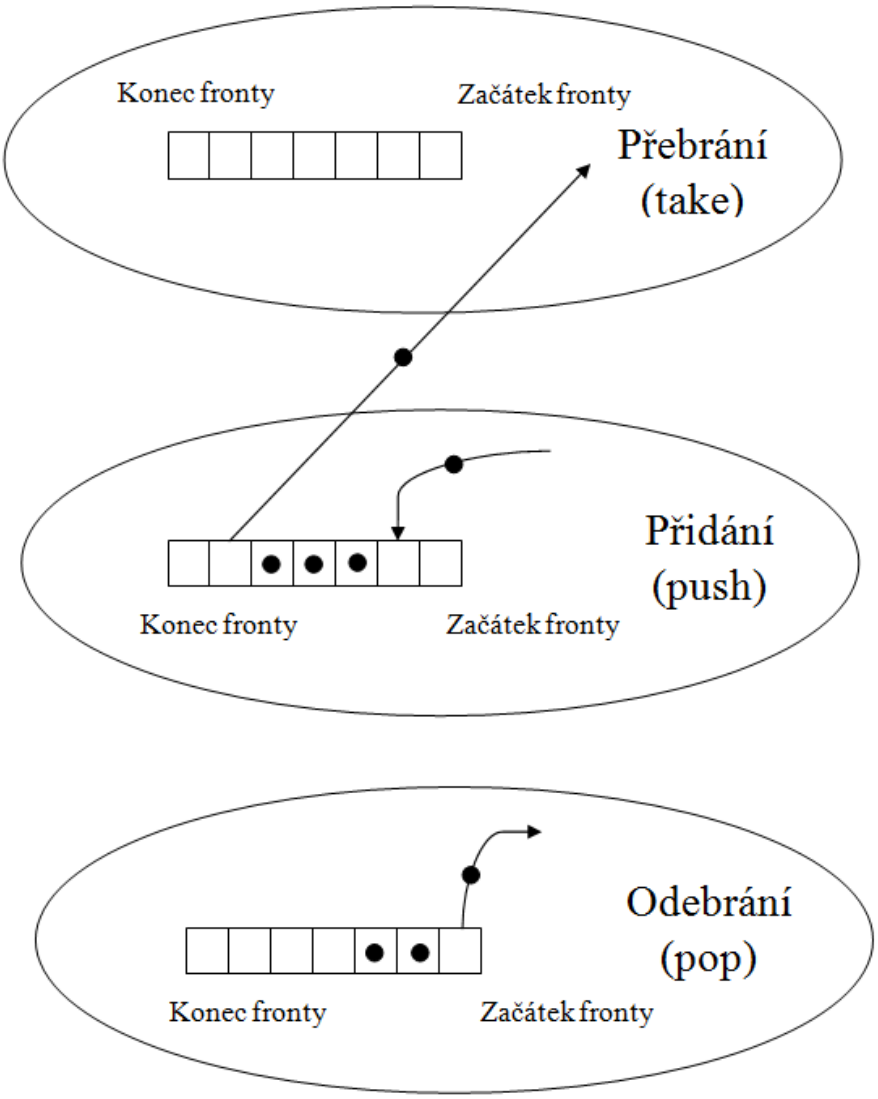
Spolupráce mezi metodami *fork()* a *join()* znázorněna graficky (viz Obrázek 1).



Obrázek 1: Spolupráce mezi fork a join metodami u instancí *ForkJoinTask*
Předloha: (Julien Ponge 2011)

2.2.3 Work-stealing technika

Fork-join framework snižuje spor mezi vlákny o pracovní frontu technikou známou jako work-stealing. Každé pracovní vlákno má svojí vlastní pracovní frontu, která je implementována pomocí oboustranné fronty. Oboustranná fronta funguje tak, že prvky mohou být přidávány nebo odebírány jak ze začátku fronty, tak z jejího konce (funkce *push* a *pop*). Když úkol vyvolá nové vlákno, umístí ho na začátek jeho vlastní fronty. Když úkol vykonává operaci *join* a čeká na jiný úkol, který ještě nebyl dokončen, místo toho aby se úkol uspal, vyzvedne další úkol ze začátku své oboustranné fronty a provede ho. V případě, že fronta úkolů vlákna je prázdná, snaží se přebrat další úkol z konce fronty jiného vlákna. Tento postup se opakuje, dokud operace *join* nedostane oznámení od úkolu pomocí metody *isDone*, že byl dokončen. (2) Operace přebírání, přidání a odebrání úkolu do oboustranné fronty jsou znázorněny na Obrázku 2.



Obrázek 2: Operace nad oboustrannou frontou

Předloha: (Doug Lea 1999)

2.3 Implementace

Celý fork/join framework byl implementován asi 800 řádky kódu v jazyce Java. Detailněji budou popsány dvě části implementace frameworku. Jedna z nich je účinná podpora operací oboustranné fronty (*push*, *pop* a *take*[VII]). Druhá z nich je řízení algoritmu *work-stealing*, pomocí něhož získávají vlákna novou práci.

[VII] Pomocí této operace může vlákno získat práci od jiného vlákna.

2.3.1 Oboustranné fronty

Základní struktura oboustranné fronty využívá jednoduché pole, spolu se dvěma indexy. *Top* (horní index) slouží jako ukazatel na první prvek pole, měnící se pomocí operací *push* a *pop*. *Base* (spodní index) se mění při použití operace *take* a ukazuje na poslední prvek pole. Vzhledem k tomu, že operace třídy *ForkJoinTask* jsou úzce vázány na konkrétní detaily oboustranné fronty, je tato datová struktura umístěna přímo v této třídě, než aby byla definována jako samostatná komponenta.

Hlavním problémem v implementaci oboustranné fronty byla synchronizace a vyhýbání se sychronizaci. Slabým místem byla především potřeba získat zámky pro každou operaci *push* a *pop*. Řešení toho problému bylo založeno na dvou myšlenkách:

- Operace *push* a *pop* jsou volány pouze z vláken, kterým patří oboustranná fronta.
- Přístup k operaci *take* může být snadno omezen pouze pro jedno vlákno současně, pomocí vstupního zámku. Tento zámek také slouží k zakázání provádění operací, když je to nutné.

Deklarování *top* a *base* indexů jako *volatile* zajišťuje, že operace *take* a *pop* mohou být provedeny bez zámků, pokud má fronta více než jeden prvek. Zjištění, zdali mohou být operace uskutečněny se provádí pomocí algoritmu, který je pro operaci *push* založen na tom, že se nejdříve dekrementuje index *top*:

if (−−top >= base) ...

a pro operaci *take* se nejdříve inkrementuje index *base*:

if (++base < top) ...

V každém případě se před provedením operací *pop* a *take* musí pomocí porovnání obou indexů zkontrolovat, jestli jejich provedení nezpůsobí, že fronta bude prázdná. Při potenciálním konfliktu mezi operacemi se používá asymetrické pravidlo: Operace *pop* překontroluje stav a snaží se pokračovat po získání zámku od fronty (stejného, který je držen operací *take*), kdy dojde k neprovedení operace pouze tehdy, pokud je fronta opravdu prázdná. U operace *take* dojde k neprovedení ihned, většinou díky tomu, že se snaží získat úkol z jiné fronty.

Použití *volatile* u indexů také umožňuje provedení operace *push* bez synchronizace, pokud by tím nepřeteklo pole fronty. V případě, že by se tak stalo, musela by být fronta uzamčena, aby mohla být změněna velikost pole. (6)

2.3.2 Přebírání úkolů a nečinnost vláken

Pracovní vlákna ve work-stealing frameworcích nijak neřeší synchronizační požadavky programů, pod kterými běží. Tato vlákna pouze vytvářejí úkoly, provádějí operace *push, pop* a *take*, spravují status úkolu a zajišťují samotné provedení úkolů. Jednoduchost tohoto systému je efektivní pouze tehdy, když je k dispozici dostatek práce pro všechna vlákna.

Hlavním problémem je to, co dělat, když pracovní vlákno nemá žádné svoje úkoly a ani nemůže žádný úkol přebrat od jiného vlákna. Pokud se vlákno bude neustále snažit přebrat úkol od jiných vláken, která nejsou nečinná, může zpomalit jejich práci.

Vlákno, kterému se nepodaří získat práci od nějakého jiného vlákna, sníží svojí prioritu předtím, než se pokusí o další přebrání. Mezi pokusy o přebrání úkolu provede *Thread.yield*[VIII] a zaregistruje se jako nečinné. Pokud se všechna ostatní vlákna stanou neaktivními, čekají na další hlavní úkoly. Jinak se vlákno po určitém počtu nepodařených pokusů dostává do fáze nečinnosti, kde je uspáno (po dobu až 100ms), mezi pokusy o přebrání úkolu. Tato uspaná vlákna mohou způsobit umělé prodlevy u programů, kterým trvá dlouho rozdělení jejich úkolů. Ale i přes tyto prodlevy se toho řešení zdá být nejlepším kompromisem. (7)

[VIII] Statická metoda, která naznačuje systému, že vlákno nemá žádnou činnost a může se tedy vzdát procesoru.

3 Doporučení pro použití knihovny

Na základě studia materiálů popisujících knihovnu fork/join bylo zpracováno několik doporučení pro zájemce, kteří by chtěli využít tuto knihovnu ve svých programech:

- Je nutné použít takový algoritmus, který může být rekurzivně rozdělen do několika nezávislých částí, kde tyto části budou provedeny samostatně (např. součet hodnot prvků v poli).
- Pokud algoritmus nevrací žádnou hodnotu je nutné navrhnout podtřídu *RecursiveAction.*
- Pokud algoritmus vrací nějakou hodnotu je nutné navrhnout podtřídu *RecursiveTask<V>*, kde V je generický parametr, který bude nahrazen datovým typem, který má být vrácen.
- Podtřída třídy *RecursiveAction* musí implementovat metodu *compute*, která bude mít návratový typ *void* a která musí obsahovat rekurzivní dělení úkolu na podúkoly a při dostatečném rozdělení přímé vyřešení problému.
- Podtřída třídy *RecursiveTask<V>* má stejné náležitosti jako podtřída třídy *RecursiveAction* s tím rozdílem, že metoda *compute* musí mít návratový typ a vracet hodnotu, kde její datový typ je určený generickým parametrem *V.*
- Rozdělení úkolu by vždy mělo být na dvě stejně velké části, kde provedení těchto částí bude zajištěno podle obecného schématu (viz. 4.1).
- Pro provedení hlavní instance podtřídy *RecursiveAction* nebo *RecursiveTask<V>* je nutné vytvořit instanci třídy *ForkJoinPool*. Samotné provedení zajistíme pomocí metody *invoke(ForkJoinTask<T> ukol)*
- Pro využití všech dostupných jader procesoru použijeme pro vytvoření instance *ForkJoinPool* konstruktor bez parametru.
- Celočíselným parametrem v konstruktoru třídy *ForkJoinPool* lze specifikovat konkrétní počet jader, které budou využívána.
- Velmi důležitým parametrem, který je třeba určit, je parametr threshold (práh). Hodnota tohoto parametru určuje, kdy bude úkol proveden přímo nebo bude rozložen na dvě menší části. Hodnota tohoto parametru není triviálně určitelná, protože vhodný threshold je pro každý algoritmus a data jiný. Nejlepším způsobem určení thresholdu je prosté zkoušení hodnot a sledování výsledků.

4 Testovací aplikace

4.1 Úvod

Pro potřeby otestování časového zefektivnění běhu aplikací při použití fork/join frameworku byl vyvinut speciální program. Tento program implementuje dva různé výpočetní algoritmy. První algoritmus na základě zvoleného obrázku zajistí jeho rozmazání (viz 6.2.1). Druhý algoritmus je čistě výpočetní, kde na základě vstupního souboru s určitým počtem molekul vody vypočte jejich párovou energii (viz 6.2.2). Vzhledem k tomu, že program je určený pro časové testy, neobsahuje žádné GUI[IX], které by mohlo ovlivnit výsledky měření. Z předcházející věty vyplývá, že program obsahuje pouze textové menu vypisované do příkazové řádky, které je však naprosto dostačující a umožní, jak ovládání programu, tak zadávání parametrů potřebných pro jeho běh.

Samotný program obsahuje celkem 15 tříd, které jsou rozděleny do 7 balíčků. Jeden balíček je zaměřen na rozhraní pro uživatele a provádění testů, tři obsahují jednotlivé implementace algoritmu pro rozmazání obrázku a další tři obsahují implementace algoritmu pro výpočet energie molekul vody. Aby bylo možné provést porovnání časů běhu programu, byly zvoleny tyto implementace:

- Sériová implementace – klasický způsob bez využití jakékoliv paralelizace.
- Implementace využívající třídu *ExecutorService* – způsob který využívá třídu, která je obsažena v Java 6 a tedy bude ilustrovat možnosti před příchodem knihovny fork/join .
- Implementace využívající fork/join framework – způsob využívající knihovnu, která je obsažená v Java 7 a je popisována v této práci.

Program má celkem čtyři módy, které mohou být voleny pomocí menu programu. V prvním módu program provádí časové testy algoritmu pro rozmazání obrázku, kde výstupem jsou dosažené časy vypsané do příkazové řádky. Druhý mód je obdoba prvního, akorát jsou prováděny časové testy algoritmu pro výpočet párové energie molekul vody. Během těchto

[IX] GUI je zkratka pro grafické uživatelské rozhraní, které umožňuje ovládat program pomocí interaktivních grafických prvků

časových testů jsou měřené časy, které reflektují délku trvání výpočetního algoritmu na základě zadaných parametrů. Těmito parametry jsou u prvního módu *blurSize*, neboli velikost rozmazání, dále *threshold*, neboli prahová hodnota a samotný obrázek, který se má rozmazat. U druhého módu je potřeba zadat pouze parametr *threshold*. Příklad prvního módu je patrný z Obrázku 3. Třetí mód po zadání stejných parametrů jako v prvním módu zajišťuje vytvoření výstupních obrázků, které reflektují funkci algoritmu pro rozmazání obrázku. Tyto obrázky budou vytvořeny do složky, která bude mít stejnou cestu jako spuštěný jar[X] soubor s programem. Poslední mód zajistí na základě zvoleného algoritmu vypočítání a vypsání celkové párové energie molekul vody.

```
C:\Users\Maxi\Downloads>java -jar BakPraceFinal-0.0.1.jar
Pocet Jader: 8
Zvolte volbu:
 1) Spustit casove testy - Rozmazani Obrazku
 2) Spustit casove testy - Vypocet energie molekul vody
 3) Ziskani rozmazanych obrazku
 4) Ziskani energie molekul vody
 5) Ukoncit program
 Volba: 1

---Zadani vstupnich parametru---

Soubor pro rozmazani:
 1) Defaultni
 2) Vlastni soubor
 Volba: 1
Pole pixelu obsahuje: 4314624 prvku

BlurSize [napr. "10,20,30"] : 20,50
Threshold pro BlurSize = 20 z rozsahu <1-4314624> :90000
Threshold pro BlurSize = 50 z rozsahu <1-4314624> :90000

Stisknete libovolnou klavesu pro spusteni testu!

        ---------- SerialBlur ----------
       Blursize          ms
             20        1305
             50        3271
        ---------- ExecutorBlur ----------
       Blursize      Threshold          ms
             20          90000         550
             50          90000        1322
        ---------- ForkjoinBlur ----------
       Blursize      Threshold          ms
             20          90000         536
             50          90000        1292

Testy probehly vporadku!
```

Obrázek 3: Příklad časových testů algoritmu pro rozmazání obrázku

Pro vývoj této aplikace bylo použité vývojové prostředí Eclipse s využitím pluginu Maven[XI]. Maven je nástroj, který usnadňuje práci při buildování aplikace.

[X] Java archive soubor, který slučuje třídy, metada a zdrojové soubory do jednoho souboru, který může být spuštěn.

[XI] http://maven.apache.org/

4.2 Výpočetní algoritmy

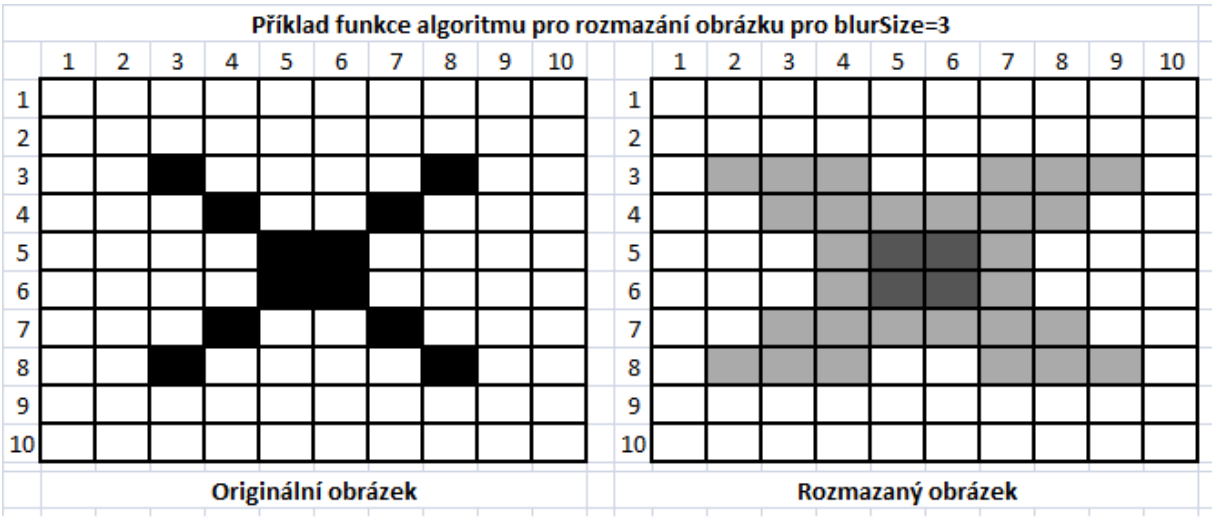
4.2.1 Rozmazání obrázku

Tento algoritmus je založen na úpravě pole pixelů, které je získáno ze vstupního obrázku, kde po úpravě hodnot jednotlivých pixelů je z toho pole vytvořen výstupní obrázek.

Každý prvek pole pixelů obsahuje osmibitovou hodnotu, do které jsou zapouzdřeny jednotlivé složky barevného modelu RGB[XII]. Z této hodnoty jsou vyextrahovány hodnoty pro jednotlivé barevné složky, tedy červenou, zelenou a modrou.

Celý algoritmus pro rozmazání obrázků je založen na následující myšlence. Po řádcích sčítáme určitý počet jednotlivých barevných složek pixelů, kde hodnoty těchto složek zprůměrujeme. Z těchto tří barevných složek vytvoříme výslednou barvu modifikovaného pixelu. Tento postup je potřeba provést pro všechny pixely obrázku. Vytvořený obrázek z tohoto pole s modifikovanými pixely bude rozmazaný. Čím větší je počet sečtených pixelů, tím větší je rozmazání obrázku. Počet sečtených pixelů je roven parametru *blurSize*. Ke každému pixelu je vždy, pokud je to možné, přičten počet pixelů nalevo od tohoto pixelu, který je roven polovině parametru *blurSize* a stejně tak je přičten stejný počet pixelů z pravé strany.

Na obrázku 4 je příklad použití algoritmu pro rozmazání obrázků. Obrázek se skládá ze 100 prvkového pole pixelů. Parametr *blurSize* je roven 3. Je jasně patrné, že na rozmazaném obrázku je barva obrazce světlejší.



Obrázek 4: Příklad rozmazání obrázku

[XII] Zkratka pro barevný model založený na adaptivním míchání červené, zelené a modré barvy

Příklad výpočtu barvy pixelu na pozici [3,3]: [XIII]

- Barevná složka R (červená): 255 + 0 + 255 = 510 / 3 = 170
- Barevná složka G (zelená): 255 + 0 + 255 = 510 / 3 = 170
- Barevná složka B (modrá): 255 + 0 + 255 = 510 / 3 = 170

4.2.2 Výpočet energie vody

Tento algoritmus vypočítává párovou energii určitého počtu molekul vody. Ve vstupním souboru máme vždy 3 atomy každé molekuly vody. Těchto molekul je buď 1000, nebo 10000. Samotný výpočet probíhá tak, že párová energie každé molekuly se skládá z dílčích energií, které vznikají mezi touto molekulou a molekulami následujícími. Celkovou energii dostaneme sečtením párových energií všech molekul.

4.3 Implementace

4.3.1 Rozmazání obrázku

Fork – join

Třída *ForkJoinWorker* reprezentuje pracovní vlákno. Tato třída je podtřídou *RecursiveAction*, z čehož plyne, že metoda *compute* nevrací žádný výsledek, respektive má návratový typ *void*. Zdrojový kód metody *compute* je následující:

```
protected void compute() {
        if((end - start) <= threshold ) {
                computeBlur();
                return;
        } else {
                int mid = (end - start) / 2;
                ForkjoinWorker fjWorker1 = new ForkjoinWorker(start, start+mid,
sourceArray, destArray, blurSize, threshold, width);
                ForkjoinWorker fjWorker2 = new ForkjoinWorker(start+mid, end,
sourceArray, destArray, blurSize, threshold, width);
                fjWorker1.fork();
                fjWorker2.compute();
                fjWorker1.join();
        }
}
```

[XIII] Bílý pixel má hodnoty RGB[255,255,255], černý pixel má hodnoty RGB[0,0,0]

Tento kód reflektuje obecný kód pro použití fork/join frameworku. Pokud je rozdíl koncového (*end*) a počátečního (*start*) indexu menší, než *threshold*, bude výpočet proveden přímo pomocí zavolání metody *computeBlur*, která obsahuje výpočetní algoritmus. Pokud je větší, určíme prostřední index, kde tato hodnota je pojmenována *mid*. Následuje vytvoření dvou pracovních vláken, kde první obsahuje první polovinu indexů, které je potřeba zpracovat a druhé obsahuje druhou polovinu. Parametry, které jsou předány pracovnímu vláknu je počáteční index, koncový index, zdrojové pole (*sourceArray*), cílové pole (*destArray*), velikost rozmazání (*blurSize*), prahová hodnota (*threshold*) a šířka řádku obrázku (*width*). Metoda *fork* zajistí asynchronní provedení prvního úkolu, pomocí metody *compute* zajistíme provedení druhého úkolu a následně pomocí metody *join* čekáme, dokud nebude první úkol dokončen.

Executor service

Hlavní funkcionalita způsobu s využitím třídy *ExecutorService* je ukryta ve třídě *ExecutorBlur*. Konkrétně metoda *runTimedTest* zajistí rozdělení práce mezi jednotlivá vlákna a jejich provedení. Zdrojový kód metody *runTimedTest* je následující:

```
public void runTimedTest() {
        int chunk = sourceArray.length / threshold;
        for(int i = 0; i < chunk; i++) {
                if (i != (chunk - 1)) {
                        execService.execute(new ExecutorWorker(i * threshold, (i + 1) *
                threshold, sourceArray, destArray, blurSize, width));
                } else {
                        execService.execute(new ExecutorWorker(i * threshold,
                        sourceArray.length, sourceArray, destArray, blurSize, width));
                }
        }
        execService.shutdown();
        while (!execService.isTerminated()) {
        }
}
```

V první řádku určíme počet kusů (*chunk*), na které bude rozdělen výpočet. Poté pomocí cyklu s pevným počtem opakování zajistíme vytvoření a provedení jednotlivých pracovních vláken, kde každé bude mít rozdílný počáteční a koncový index. Tyto pracovní vlákna jsou implementací rozhraní *Runnable,*tedy provádějí metodu *run*. Tato metoda provádí metodu *computeBlur* a její návratový typ je *void*. Poté již jen čekáme na provedení všech pracovních vláken. Na rozdíl od způsobu využívajícího fork/join framework zde není použita žádná rekurze, ale přímé rozdělení práce.

Algoritmus pro rozmazání obrázku

```
public void computeBlur() {
        int sidePixels = (blurSize - 1) / 2;
        for(int i = start; i < end;i++) {
                int r = 0, g = 0, b = 0, count = 0;
                int rowNum = i / width;
                for(int j = -sidePixels; j <= sidePixels; j++) {
                        int index = Math.min(Math.max(0, j+i), sourceArray.length-1);
                        if(index >= (rowNum * width) && index <= ((rowNum + 1) * width))
                        {
                                int pixel = sourceArray[index];
                                Color color = new Color(pixel);
                                r += color.getRed();
                                g += color.getGreen();
                                b += color.getBlue();
                                count++;
                        }
                }
                destArray[i] = new Color((int)(r / count),(int)(g / count),(int)(b /
                count)).getRGB();
        }
}
```

V prvním řádku je parametr *blurSize* (velikost rozmazání) snížen o 1 a vydělen dvěma. Tím zjistíme kolik bude přičteno pixelů z levé a pravé strany, kde tato hodnota je pojmenována jako *sidePixels*. Dále budeme procházet všechny prvky mezi indexy *start* až *end*. Deklarujeme celočíselné proměnné *r*, *g*, *b*. Tyto proměnné budou použity pro sečtení barevných složek jednotlivých pixelů. Dále deklarujeme pomocnou celočíselnou proměnnou *count*, která bude obsahovat počet pixelů, které byly sečteny a proměnnou *rowNum*, která obsahuje číslo řádku, na kterém se nachází aktuálně upravovaný pixel. Tato hodnota je získána vydělením konkrétního indexu v poli s velikostí jednoho řádku (*width*). Následuje cyklus, ve kterém bude proveden samotný výpočet. Nejprve je třeba zajistit, abychom se s indexem nedostali mimo hranice pole. Toto je zajištěno pomocí dvou funkcí ze třídy *Math*[XIV]. Aby index nebyl menší, než je spodní hranice pole zajistíme pomocí funkce, která vrátí větší hodnotu, kde jejími parametry je 0 a index. A na druhou stranu, aby index nebyl větší, než je poslední index pole využíváme funkci, která vrátí menší hodnotu z indexu a posledního indexu pole. Následuje podmínka zajišťující, aby nebyly přičteny indexy, které nejsou na stejné řádce. Poté již probíhá samotný výpočet, kdy si nejdříve získáme pixel z pole, poté pomocí třídy *Color* a pixelu vyextrahujeme jeho barevné složky a přičteme je

[XIV] Statická třída obsahující metody pro základní matematické operace.

k odpovídajícím proměnným. Po sečtení všech prvků vytvoříme v cílovém poli modifikovaný pixel. K tomu opět využijeme třídu *Color*, kde jednotlivé barevné složky v konstruktoru vydělíme počtem pixelů, které byly sečteny a přetypujeme je na celá čísla.

4.3.2 Výpočet energie vody

Algoritmus

Algoritmus pro výpočet párové energie mezi molekulami vody je poměrně složitý a byl převzat ze zdrojového kódu programu[XV], který byl napsán v jazyce Fortran RNDr. Milanem Předotou, Ph.D.

Fork – join

Třída *ForkJoinWorker* je zde na rozdíl od předchozího případu podtřídou *RecursiveTask<Double>* což znamená, že metoda *compute* má návratový typ *double*.

Zdrojový kód metody *compute* je následující:

```
protected Double compute() {
        if((end - start) < threshold) {
                for(int i = start; i < end; i++) {
                        for(int j=i+1; j < nmol; j++) {
                                energy += energy12(i, j);
                        }
                }
        } else {
                int mid = (end - start) / 2;
                ForkjoinWorker fjWorker1 = new ForkjoinWorker(r, nmol, L, start, start +
mid, threshold);
                ForkjoinWorker fjWorker2 = new ForkjoinWorker(r, nmol, L, start + mid, end,
threshold);
                fjWorker1.fork();
                energy = fjWorker2.compute() + fjWorker1.join();
}
 return energy;
}
```

Implementace této metody v podstatě reflektuje předchozí implementaci u algoritmu pro rozmazání obrázků. Pokud je rozdíl koncového (*end*) a počátečního (*start*) indexu menší, než *threshold* neboli prahová hodnota, bude proveden výpočet energie molekul mezi těmito

[XV] http://ufy.prf.jcu.cz/vyuka/upp/water/water_serial.f90

indexy. Pokud je větší, bude počet indexů rozdělen na polovinu a každá polovina bude zpracována jedním pracovním vláknem. Pracovním vláknům je pomocí konstruktoru předáváno pole s molekulami (*r*), počet molekul (*nmol*), velikost boxu (*L*) a počáteční a koncový index. Metoda *fork* zajistí asynchronní provedení prvního úkolu, pomocí metody *compute* zajistíme provedení druhého úkolu a následně pomocí metody *join* čekáme, až první úkol bude dokončen a vrátí výsledek. Zde výsledek bude typu *double*, protože pracovní vlákna jsou podtřídou třídy *RecusiveTask<double>*. Tato hodnota představuje párovou energii molekul vody mezi počátečním a koncovým indexem a je návratovým parametrem metody *compute*.

Executor Service

Hlavní funkcionalita způsobu s využitím třídy *ExecutorService* je ukryta ve třídě *ExecutorCompute*. Konkrétně metoda *runTimedTest* zajistí rozdělení práce mezi jednotlivá vlákna a jejich provedení. Zdrojový kód metody *runTimedTest* je následující:

```
public double runTimedTest() {
	int chunk = nmol / threshold;
	for (int i = 0; i < chunk; i++) {
		if (i != (chunk - 1)) {
			Future<Double> future = execService.submit(new ExecutorWorker(r,
nmol, L, i * threshold, (i + 1) * threshold));
			futureSet.add(future);
		} else {
			Future<Double> future = execService.submit(new ExecutorWorker(r,
nmol, L, i * threshold, nmol + 1));
			futureSet.add(future);
		}
	}
	for (Future<Double> future : futureSet) {
		try {
			sum += future.get();
		} catch (InterruptedException e) {
			System.err.println("Nekde se stala chyba! \n Program bude ukoncen");
			System.exit(0);
		} catch (ExecutionException e) {
			System.err.println("Nekde se stala chyba! \n Program bude ukoncen");
			System.exit(0);
		}
	}
	ExecService.shutdown();
	return sum;
}
```

V první řádku určíme počet kusů (*chunk*), na které bude rozdělen výpočet. Tato hodnota je dána podílem počtu molekul a prahovou hodnotou. Pomocí cyklu s pevným počtem opakování je rozdělena práce mezi pracovní vlákna a jejich provedení je zajištěno pomocí metody *submit*. Pracovním vláknům je pomocí konstruktoru předáváno pole s molekulami (*r*), počet molekul (*nmol*), velikost boxu (*L*) a počáteční a koncový index. Pracovní vlákno *ExecutorWorker* implementuje rozhraní *Callable<Double>*, aby bylo možno získat výsledek dílčího výpočtu. Výsledky jednotlivých pracovních vláken jsou instancemi třídy *Future<Double>*. Tyto jednotlivé instance jsou uloženy do množiny typu *HashSet*. Získání výsledku ze třídy *Future* pro všechna pracovní vlákna je provedeno pomocí metody *get*, kde tyto výsledky jsou sečteny a celkový výsledek je vrácen jako návratová hodnota metody *runTimedTest*.

5 Experiment

Jedním z hlavních cílů této práce bylo na základě vhodně zvolených algoritmů ověřit časovou úsporu běhu aplikací při použití fork/join frameworku. Experiment probíhal s testovací aplikací, která je popsaná v kapitole 6. Podmínky, za kterých byly testy prováděny jsou detailně popsané v testovací metodice (viz. 3.1).

Při provádění experimentů bylo celkem naměřeno 1200 hodnot. Jednotlivé výsledky roztřízené celkem do 10 tabulek jsou k dispozici v příloze. Tyto hodnoty byly zprůměrovány, byla vypočtena směrodatná odchylka a zrychlení. Zrychlení bylo zaneseno do grafů (viz 3.1).

Z grafu č. 1 je patrné, že se zvětšujícím se počtem jader roste zrychlení téměř lineárně. Na 2 jádrovém procesoru je algoritmus využívající fork/join framework přibližně 1,5x rychlejší než sériová verze tohoto algoritmu. Z toho vyplývá, že na tomto počítači ušetří fork/join framework téměř polovinu času, který je potřeba na provedení sériové verze algoritmu pro rozmazání obrázku. Dále je z grafu patrné, že různé velikosti rozmazání obrázku výsledné zrychlení ovlivňují jen minimálně.

Graf č. 2 obsahuje hodnoty, které byly naměřeny se stejnými parametry jako hodnoty z grafu č. 1 ale operačním systémem byl Linux. Do tohoto grafu je také zahrnuto zrychlení, které bylo vypočteno z naměřených hodnot na výpočetních clusterech (16 a 24 jader). Z grafu lze vyčíst, že hodnota zrychlení na výpočetním clusteru je mnohem větší, než na běžných počítačích. Porovnáním zrychlení mezi platformou Linux a Windows zjistíme, že se hodnoty v podstatě rovnají.

Graf č. 3 a graf č. 4 reprezentují zrychlení při využití fork/join frameworku oproti ExecutorService na platformách Windows a Linux. Z obou grafů lze vyčíst, že rozdílné velikosti rozmazání obrázku ovlivňují výsledné zrychlení jen minimálně. Na počítačích s Linuxem (graf č. 4) je zrychlení při využití fork/join frameworku oproti *ExecutorService* do 10%, kde tento trend narušuje pouze výpočetní cluster, na kterém bylo zrychlení kolem 20%. Na platformě Windows se zrychlení se zvyšujícím se počtem jader snižovalo. Na 8 jádru se zrychlení blížilo hodnotě 1, čili zrychlení bylo jen minimální.

Z Grafu č. 5 a č. 6 je patrné, že se zvyšujícím se počtem jader roste zrychlení opět téměř lineárně. Tím bylo prokázáno, že rozdílný algoritmus nemá na růst zrychlení téměř žádný vliv. Dále vidíme, že se zvyšujícím se počtem jader se zvyšuje rozdíl ve zrychlení mezi různými počty molekul. Tento jev je dán dobou výpočtu obou algoritmů. Na 24 resp. 16

jádrovém stroji trval v průměru výpočet pro 1000 molekul při použití knihovny fork/join pouze 81 resp. 116 ms. Tento výpočet je příliš rychlý na to, aby se mohl pořádně projevit přínos architektury s 16 resp. 24 jádry. Při výpočtu energie pro 10000 molekul bylo na 24 jádru dosaženo zrychlení 11.

Posledními grafy č. 7 a č. 8 je zrychlení programu pro výpočet energie molekul vody při využití fork/join frameworku oproti ExecutorService na platformách Windows a Linux. Z obou grafů je patrné, že zrychlení je téměř totožné a se zvětšujícím se počtem jader procesoru se téměř nemění. Různý počet molekul toto zrychlení také neovlivňuje. Z obou grafů lze vyvodit závěr, že zrychlení fork/join frameworku je přibližně 20% a to jak na platformě Windows, tak Linux. Z toho lze usoudit, že operační systém zrychlení téměř neovlivňuje.

Kromě průměrných hodnot trvání výpočtu pro každou implementaci a parametr byly také vypočteny směrodatné odchylky, které reprezentují, jak moc se naměřené hodnoty od sebe liší. Z těchto hodnot bylo vyvozeno, že čím delší je výpočet, tím více se naměřené hodnoty od sebe liší. Například z tabulky č. 6 lze vyčíst, že pro 10000 molekul trval běh sériového výpočtu na platformě Windows průměrně 68 sekund, kde směrodatná odchylka je rovna téměř 2 sekundám. Z této hodnoty plyne, že rozptýlení naměřených hodnot bylo značné. Toto rozptýlení mohlo být způsobené tím, že po dobu takto dlouhého výpočtu mohl procesor počítače provádět i jiné činnosti, než jen samotný výpočet. Naproti tomu z tabulky č. 7 vidíme, že směrodatná odchylka pro sériový výpočet energie vody pro 1000 molekul je přibližně rovna 3 ms. Z toho plyne, že naměřené hodnoty se od sebe jen velmi málo liší.

Současně s časovými testy proběhlo také měření vytížení systémových prostředků. Výsledky byly prezentovány graficky a jsou součástí přiloženého CD (viz. Příloha A). Obrázky obsahují celkem 3 grafy. První reflektuje procentuální vytížení procesoru. Druhý velikost alokované a použité haldy[XVI]. Třetí graf obsahuje počet žijících vláken. Z grafů vytížení procesoru je patrné, že sériová verze využívá vždy pouze jedno jádro a tedy přibližné procentuální využití procesoru může být vypočteno podle vztahu *1 / počet_jader * 100* [%]. Naproti tomu, obě paralelní verze využívají vždy maximální výpočetní výkon procesoru. Z grafů naměřených pro algoritmus výpočtu energie molekul vody, který využívá třídu ExecutorService je patrné, že vytížení procesoru má velké výkyvy. Například v 7 měření na 4 jádrovém procesoru na platformě Windows (viz. Příloha A) je vteřinový úsek, kdy bylo vytíženo pouze 1 jádro. Při využití fork/join frameworku tento jev nenastal. Tímto bylo prakticky ověřeno, že fork/join framework přináší algoritmus, který dokáže efektivněji využít

[XVI] Paměť, která uchovává všechny programem vytvořené objekty.

paralelní architekturu. Lepším využitím paralelní architektury je myšleno, že práce je lépe rozdělena mezi vlákna.

Další zajímavé zjištění plyne z grafu žijících vláken. Zatímco při využití třídy ExecutorService je počet žijících vláken roven počtu, který vzniká podílem celkového počtu prvků s prahovou hodnotou (threshold). Při využití fork/join frameworku je počet žijících vláken roven vždy pouze počtu jader procesoru. Tento jev je způsoben tím, že u fork/join frameworku jsou žijící vždy pouze pracovní vlákna, kde jejich počet je roven počtu jader procesorů. Každé takové pracovní vlákno má svojí oboustrannou frontu (viz. 4.3.1) v které má uloženy podúkoly, které postupně provádí.

Vzhledem k tomu, že testy byly prováděny na různých počítačích s různými procesory a tedy i s různými frekvencemi procesoru, nelze mezi sebou přímo porovnávat naměřené hodnoty. Nicméně cílem práce bylo prokázat časovou úsporu, tedy zrychlení.

6 Závěr

Závěrem lze říci, že práce splnila všechny vytyčené cíle. Vhodnou kompozicí dostupné literatury vznikl popis knihovny, který slouží pro pochopení návrhu a funkce fork/join frameworku. Vznikla přesná doporučení pro zájemce, kteří by chtěli využít fork/join framework ve svých aplikacích, ale nemají s ním žádné zkušenosti.

Samostatnou kapitolou bylo prokázání časové efektivity při nasazení fork/join frameworku. Byly zvoleny dva algoritmy, na kterých byla prezentována časová úspora při využití fork/join frameworku. Testy byly prováděny na osobních počítačích s takovými počty jader, které jsou dnes nejrozšířenější (2, 4 a 8 jader). Experimentálně byly testy provedeny také na výpočetním clusteru (16 a 24 jader). Díky tomu byl otestován přínos knihovny nejen na běžných osobních počítačích, ale také na architektuře, která se zabývá striktně výpočty. Bylo jednoznačně prokázáno, že fork/join framework přináší časovou úsporu, která je úměrná počtu jader procesoru. Dále bylo zjištěno, že operační systém, na kterém je program spuštěn nemá na zrychlení téměř žádný vliv.

V současnosti se vícejádrové procesory vyskytují téměř v každém novém počítači, ale jen velmi málo aplikací je efektivně využívá. Tato práce přinesla popis nového frameworku pomocí kterého lze vícejádrovou architekturu efektivně a relativně snadno využít.

7 Bibliografie

1. PONGE, Julien. Java Can Excel at Painless Parallel Programming Too!. *Oracle Technology Network* [online]. 2011[cit. 2012-02-03]. Dostupné z: http://www.oracle.com/technetwork/articles/java/fork-join-422606.html

2. GOETZ, Brian. Java theory and practice: Stick a fork in it. *DeveloperWorks* [online]. 2007[cit. 2012-02-03]. Dostupné z: https://www.ibm.com/developerworks/java/library/j-jtp11137/

3. LEA, Doug. *A Java Fork/Join Framework* [online]. 1999, s. 1-2 [cit. 2012-02-03]. Dostupné z: http://gee.cs.oswego.edu/dl/papers/fj.pdf

4. Class ForkJoinPool. *Java SE7 Documentation* [online]. 2011[cit. 2012-02-03]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html

5. Class ForkJoinTask<V>. In: *Java SE7 Documentation* [online]. 2011[cit. 2012-02-03]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html

6. LEA, Doug. *A Java Fork/Join Framework* [online]. 1999, s. 3-4 [cit. 2012-02-03]. Dostupné z: http://gee.cs.oswego.edu/dl/papers/fj.pdf

7. LEA, Doug. *A Java Fork/Join Framework* [online]. 1999, s. 4 [cit. 2012-02-03]. Dostupné z: http://gee.cs.oswego.edu/dl/papers/fj.pdf

8 Seznam příloh

PŘÍLOHA A: Obsah přiloženého CD

PŘÍLOHA B: Grafy zrychlení pro algoritmus provádějící rozmazání obrázku

PŘÍLOHA C: Grafy zrychlení pro algoritmus provádějící výpočet energie molekul vody

PŘÍLOHA D: Výsledky experimentu pro algoritmus provádějící rozmazání obrázku

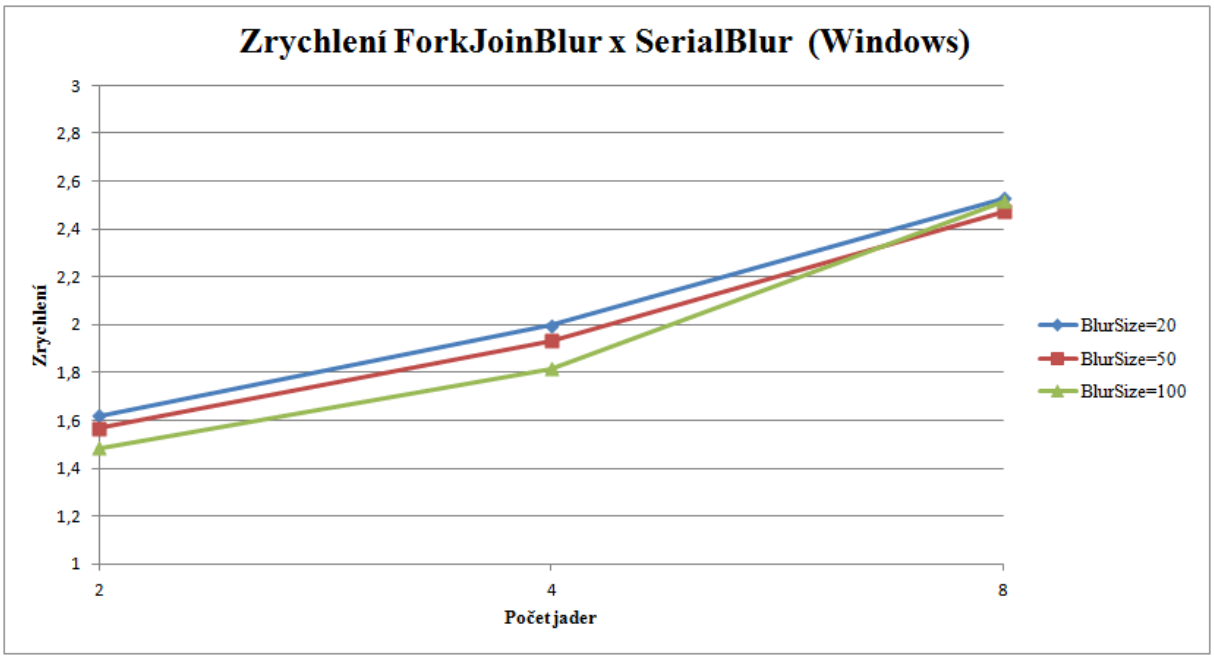
PŘÍLOHA E: Výsledky experimentu pro algoritmus provádějící výpočet energie molekul vody

Příloha A

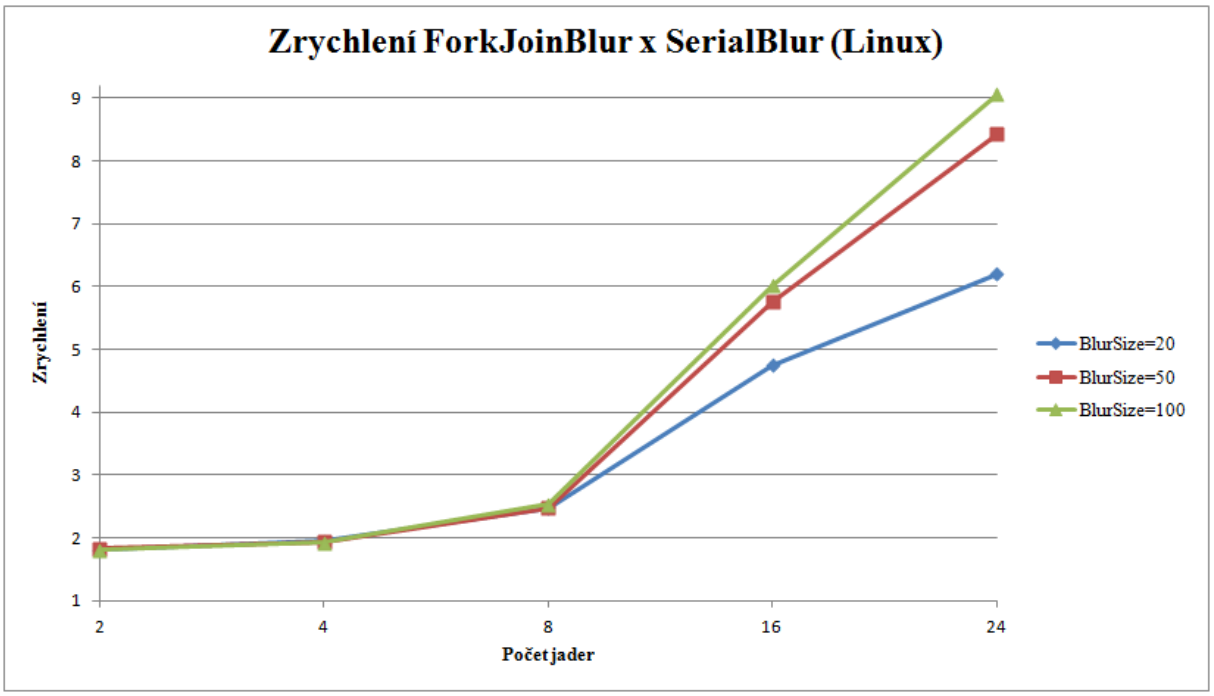
- Adresář *TestApp* se zdrojovým kódem programu
- Adresář *Dokumentace_RozmazaniObrazku* obsahující 2 podadresáře:
    - *vysledky_mereni* – obsahuje vyfocené výsledky pro každý test
    - *vytizeni_prostredku* – obsahuje obrázky vytížení systémových prostředků
- Adresář *Dokumentace_VypocetEnergie* obsahující 2 podadresáře:
    - *vysledky_mereni* – obsahuje vyfocené výsledky pro každý test
    - *vytizeni_prostredku* – obsahuje obrázky vytížení systémových prostředků
- Spustitelný jar soubor s testovacím programem
- Text bakalářské práce *Testování knihovny fork/join v jazyce Java 7* ve formátech pdf, doc a docx

Příloha B

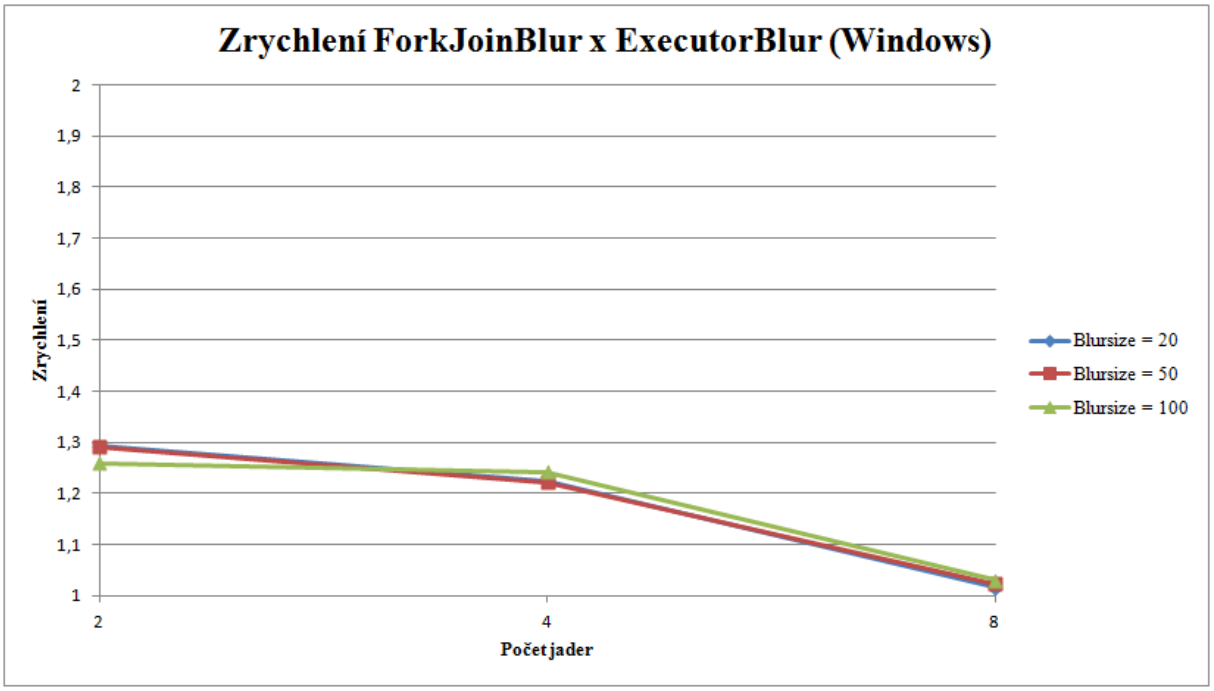
Graf č. 1 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti sériovému programu na platformě Windows.



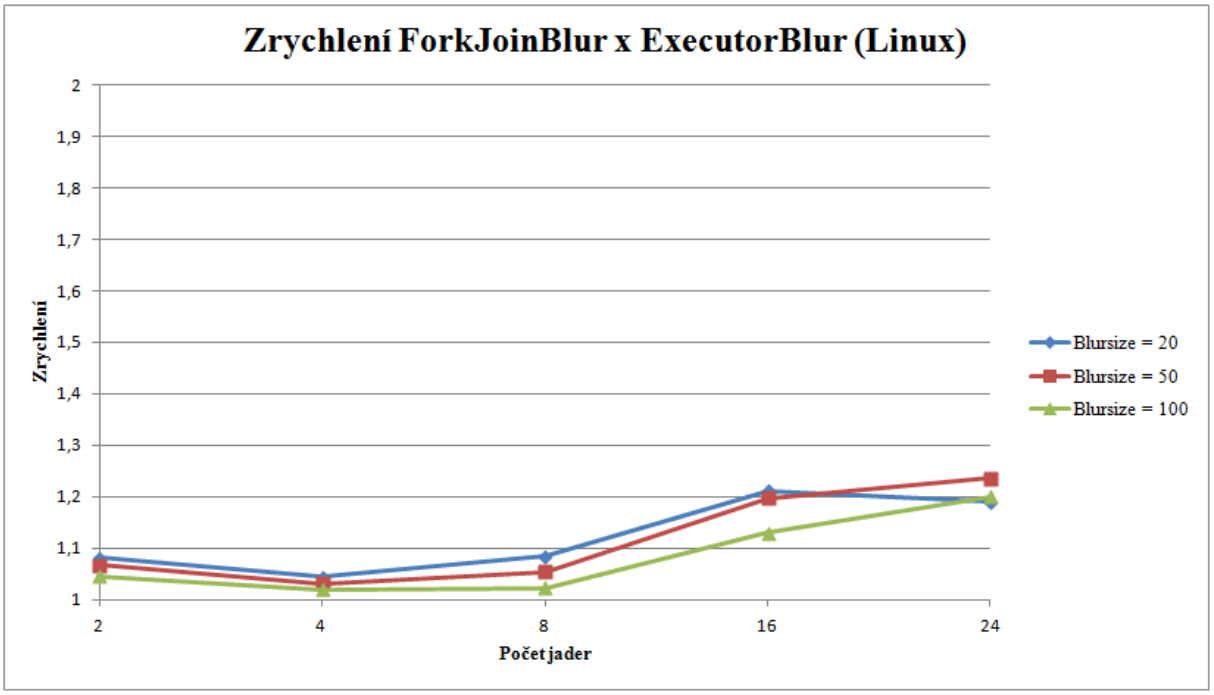
Graf č. 2 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti sériovému programu na platformě Linux.



Graf č. 3 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti ExecutorService na platformě Windows.

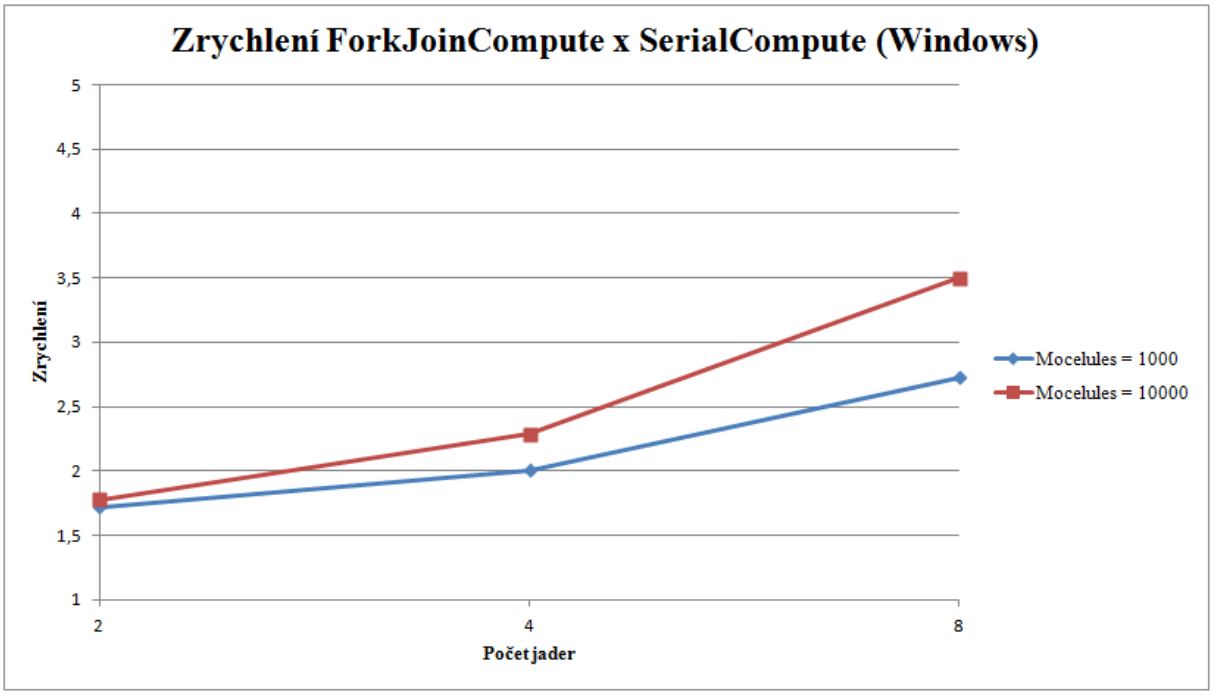


Graf č. 4 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti ExecutorService na platformě Linux.

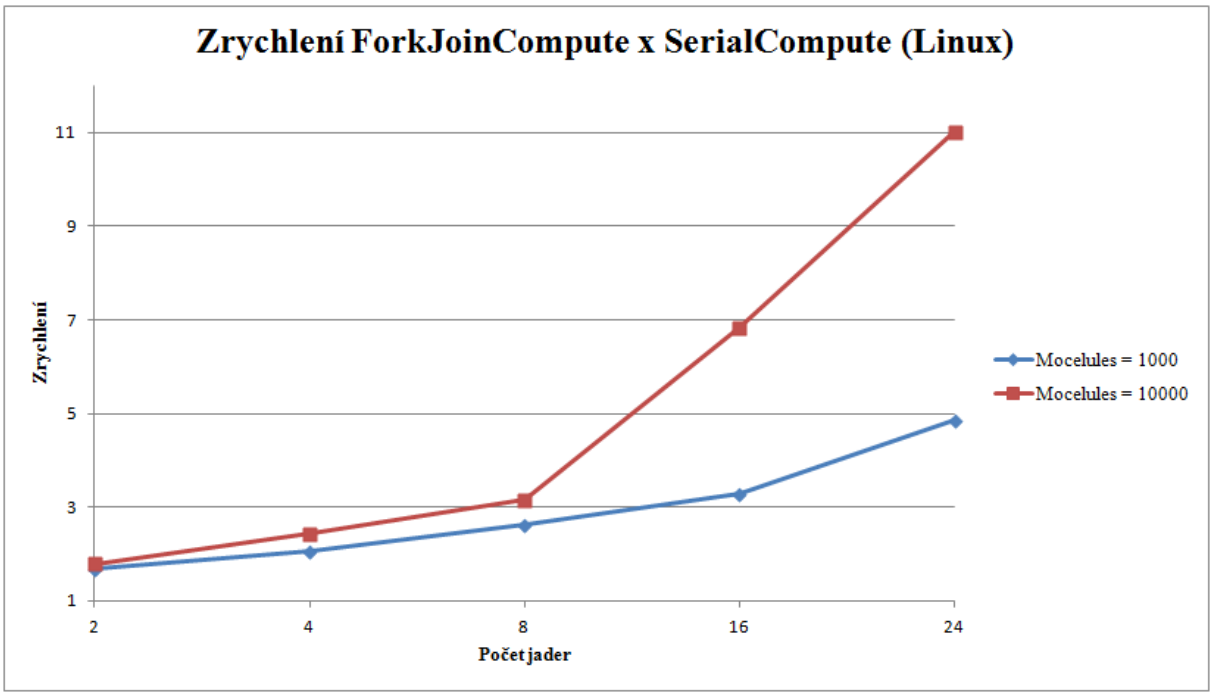


Příloha C

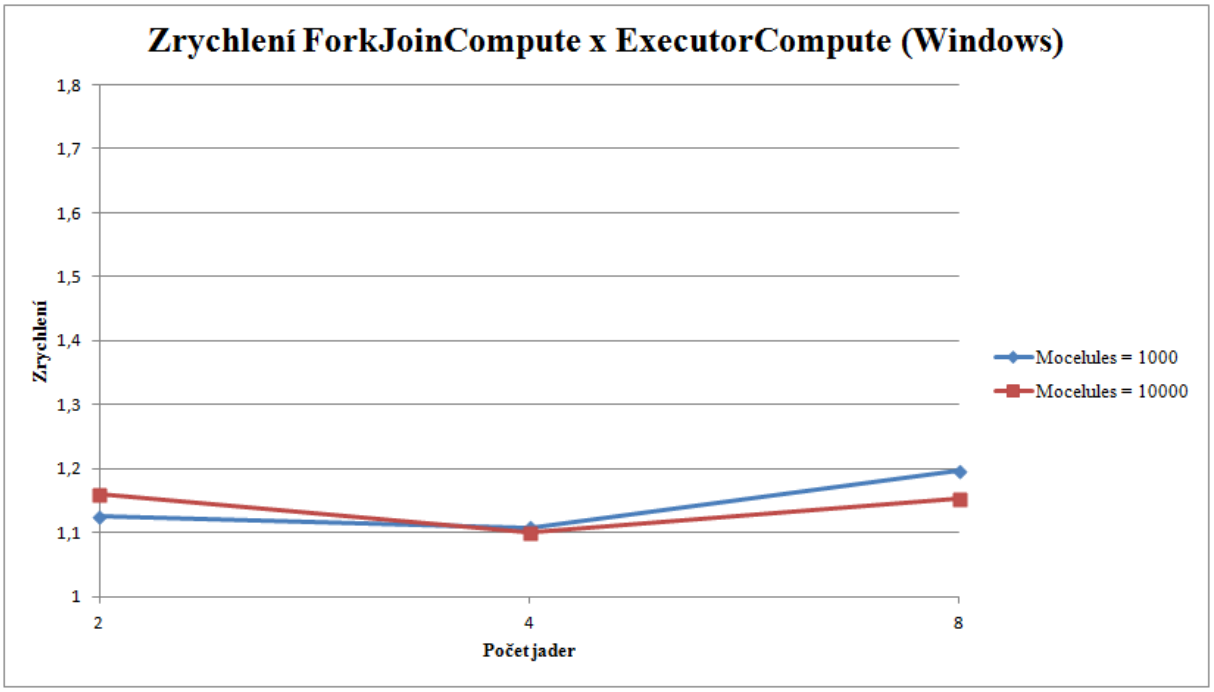
Graf č. 5 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti sériovému programu na platformě Windows.



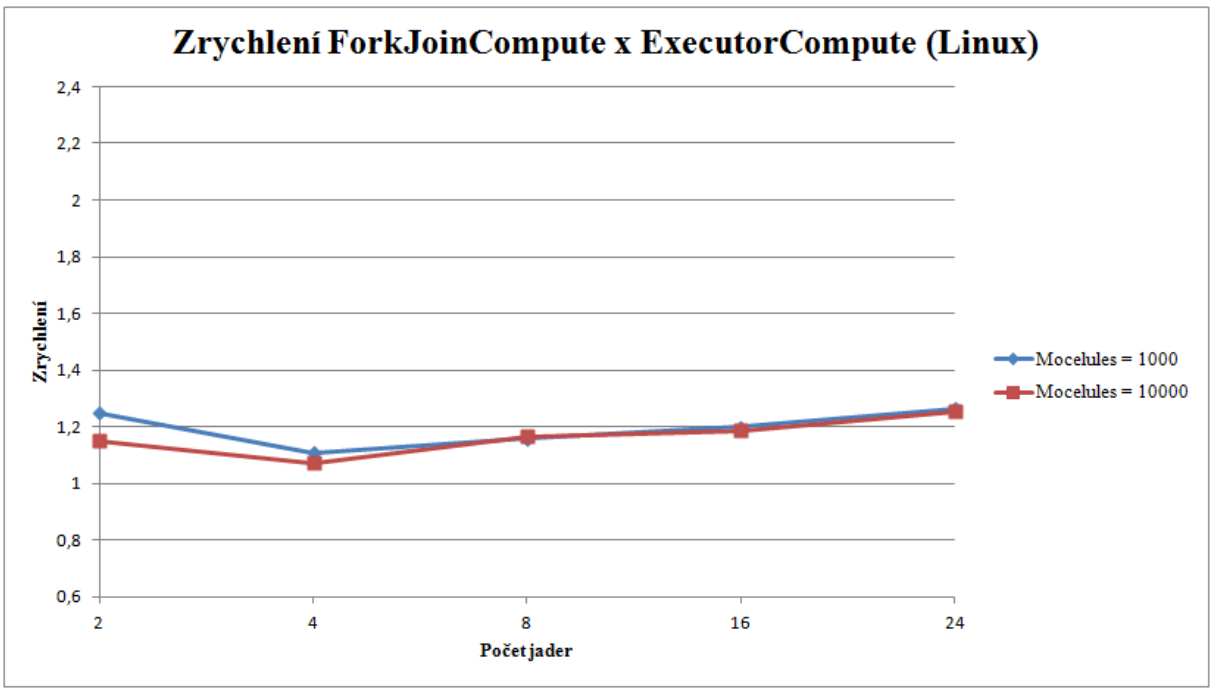
Graf č. 6 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti sériovému programu na platformě Linux.



Graf č. 7 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti třídě ExecutorService na platformě Windows.



Graf č. 8 reprezentující zrychlení běhu programu při využití forkjoin frameworku oproti třídě ExecutorService na platformě Linux.



Příloha C

Tabulka č. 1 s naměřenými výsledky běhu programu pro 2 jádrový procesor.

Číslo	SerialBlur		ExecutorBlur		ForkjoinBlur	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
BlurSize: 20 Threshold: 90000						
1	3391	2670	2754	1652	1974	1466
2	3195	2754	2606	1618	1978	1497
3	3144	2664	2643	1632	2023	1598
4	3221	2767	2392	1532	2015	1506
5	3224	2683	2375	1703	1983	1448
6	3215	2729	2520	1543	2038	1488
7	3206	2767	2563	1568	1992	1445
8	3186	2658	2541	1564	1964	1505
9	3186	2721	2929	1688	1964	1567
10	3257	2720	2397	1698	1973	1460
Průměr	**3222,5**	**2713,3**	**2572**	**1619,8**	**1990,4**	**1498**
σ	**62,85**	**40,07**	**165,03**	**61,87**	**24,72**	**47,7**
BlurSize: 50 Threshold: 90000						
1	8136	6698	6490	3906	4870	3603
2	7466	6760	6295	3794	4874	3712
3	7636	6760	7111	3875	4987	3709
4	7905	6655	6202	3992	4963	3704
5	7652	6692	5888	3938	4934	3622
6	7658	6814	6206	3940	4925	3680
7	7598	6727	6000	3999	4879	3711
8	7400	6704	6086	3987	4864	3618
9	7665	6811	6848	3782	4843	3705
10	7708	6755	6114	3947	4843	3609
Průměr	**7682,4**	**6737,6**	**6324**	**3916**	**4898,2**	**3667,3**
σ	**198,40**	**49,24**	**367,01**	**73,72**	**48,06**	**45,38**
BlurSize: 100 Threshold: 90000						
1	14584	13243	11716	7682	9599	7312
2	14316	13223	12798	7702	9608	7404
3	14231	13447	12995	7699	9900	7440
4	14351	13222	12423	7801	9817	7362
5	14348	13175	11385	7648	9607	7320
6	14358	13423	11889	7757	9695	7380
7	14393	13274	12030	7712	9611	7351
8	14116	13210	13130	7678	9621	7299
9	14365	13287	12368	7752	9575	7445
10	14396	13367	10902	7620	9523	7338
Průměr	**14345,8**	**13287,1**	**12163,6**	**7705,1**	**9655,6**	**7365,1**
σ	**113,54**	**89,10**	**680,30**	**50,95**	**110,65**	**48,88**

Tabulka č. 2 s naměřenými výsledky běhu programu pro 4 jádrový procesor.

Číslo	SerialBlur		ExecutorBlur		ForkjoinBlur	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
BlurSize: 20 Threshold: 90000						
1	1321	1043	782	548	680	540
2	1344	1039	816	540	649	524
3	1319	1041	841	558	675	537
4	1304	1043	783	553	637	531
5	1305	1029	821	550	691	535
6	1311	1029	824	571	660	529
7	1312	1037	805	579	657	541
8	1308	1041	830	572	640	533
9	1306	1042	787	548	641	525
10	1312	1048	780	563	654	550
Průměr	**1314,2**	**1039,2**	**806,9**	**558,2**	**658,4**	**534,5**
σ	**11,28**	**5,78**	**21,41**	**12,02**	**17,37**	**7,51**
BlurSize: 50 Threshold: 90000						
1	3155	2581	1950	1380	1627	1316
2	3269	2549	2024	1396	1593	1330
3	3076	2588	1869	1364	1617	1323
4	3076	2589	1964	1388	1615	1358
5	3105	2554	1920	1348	1581	1313
6	3075	2587	1973	1365	1605	1341
7	3053	2569	1911	1371	1635	1320
8	3076	2558	1905	1379	1559	1316
9	3066	2568	1973	1353	1613	1350
10	3078	2596	2127	1366	1604	1325
Průměr	**3102,9**	**2573,9**	**1961,6**	**1371**	**1604,9**	**1329,2**
σ	**61,42**	**15,7**	**69,08**	**14,22**	**21,28**	**14,68**
BlurSize: 100 Threshold: 90000						
1	5678	5098	3822	2692	3118	2634
2	5858	5061	3810	2714	3083	2625
3	5663	5067	3850	2690	3275	2654
4	5590	5103	3826	2668	3061	2624
5	5571	5072	3784	2698	3077	2654
6	5618	5073	3860	2713	3099	2615
7	5574	5099	3890	2678	3137	2656
8	5620	5064	3844	2682	3087	2621
9	5630	5101	3844	2682	3072	2652
10	5600	5079	4056	2683	3066	2648
Průměr	**5640,2**	**5081,7**	**3858,6**	**2690**	**3107,5**	**2638,3**
σ	**79,81**	**15,89**	**71,26**	**14,06**	**60,17**	**15,27**

Tabulka č. 3 s naměřenými výsledky běhu programu pro 8 jádrový procesor.

Číslo	SerialBlur		ExecutorBlur		ForkjoinBlur	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
BlurSize: 20 Threshold: 90000						
1	1413	1320	560	573	533	564
2	1380	1395	559	621	546	550
3	1454	1311	560	574	539	542
4	1333	1323	544	607	552	531
5	1339	1317	567	572	514	538
6	1340	1330	541	613	541	516
7	1353	1338	537	557	513	537
8	1369	1309	534	577	530	532
9	1306	1320	533	553	579	517
10	1312	1305	529	572	530	541
Průměr	**1359,9**	**1326,8**	**546,4**	**581,9**	**537,7**	**536,8**
σ	**43,55**	**24,56**	**13,09**	**22,23**	**18,18**	**13,57**
BlurSize: 50 Threshold: 90000						
1	3251	3327	1307	1347	1329	1343
2	3228	3260	1329	1368	1318	1356
3	3276	3353	1357	1417	1298	1284
4	3315	3221	1330	1411	1290	1321
5	3262	3292	1377	1387	1306	1299
6	3215	3274	1379	1404	1367	1339
7	3230	3210	1396	1348	1300	1299
8	3234	3324	1324	1424	1320	1343
9	3264	3235	1281	1478	1296	1327
10	3231	3271	1355	1362	1313	1309
Průměr	**3250,6**	**3276,7**	**1343,5**	**1394,6**	**1313,7**	**1322**
σ	**28,22**	**45,20**	**33,86**	**38,66**	**21,22**	**22,41**
BlurSize: 100 Threshold: 90000						
1	6675	6662	2732	2715	2546	2568
2	6632	6665	2628	2629	2595	2620
3	6620	6480	2691	2621	2529	2628
4	6481	6710	2646	2656	2617	2571
5	6454	6634	2642	2628	2589	2570
6	6425	6443	2737	2690	2605	2543
7	6456	6429	2722	2641	2593	2553
8	6549	6397	2680	2639	2555	2618
9	6540	6547	2675	2627	2651	2638
10	6460	6594	2567	2614	2672	2552
Průměr	**6529,2**	**6556,1**	**2672**	**2646**	**2595,2**	**2586,1**
σ	**83,40**	**106,96**	**50,30**	**30,81**	**42,48**	**33,98**

Tabulka č. 4 s naměřenými výsledky běhu programu pro 16 jádrový procesor.

Číslo	SerialBlur	ExecutorBlur	ForkjoinBlur
	Linux [ms]	**Linux [ms]**	**Linux [ms]**
BlurSize: 20 Threshold: 90000			
1	1203	271	259
2	1195	283	219
3	1199	331	211
4	1199	343	235
5	1199	391	255
6	1179	203	287
7	1199	303	291
8	1195	303	263
9	1199	355	207
10	1203	267	291
Průměr	**1197**	**305**	**251,8**
σ	**6,51**	**50,51**	**30,87**
BlurSize: 50 Threshold: 110000			
1	2991	587	543
2	2979	615	459
3	3007	663	451
4	2987	483	471
5	3003	635	603
6	2983	655	575
7	2995	599	543
8	2987	663	559
9	2979	655	511
10	2991	659	475
Průměr	**2990,2**	**621,4**	**519**
σ	**8,91**	**53,10**	**50,47**
BlurSize: 100 Threshold: 120000			
1	5951	935	919
2	5935	1007	939
3	5955	955	831
4	5959	1743	1119
5	5967	943	927
6	5975	1355	1087
7	5955	1075	1099
8	5939	1143	967
9	5935	971	963
10	5967	1059	1043
Průměr	**5953,8**	**1118,6**	**989,4**
σ	**13,27**	**240,28**	**88,82**

Tabulka č. 5 s naměřenými výsledky běhu programu pro 24 jádrový procesor.

Číslo	SerialBlur	ExecutorBlur	ForkjoinBlur
	Linux [ms]	**Linux [ms]**	**Linux [ms]**
BlurSize: 20 Threshold: 90000			
1	1021	169	141
2	1012	182	154
3	998	183	153
4	1012	286	193
5	995	201	162
6	1005	219	168
7	994	178	173
8	1030	148	154
9	994	195	172
10	983	166	150
Průměr	**1004,4**	**192,7**	**162**
σ	**13,60**	**36,29**	**14,18**
BlurSize: 50 Threshold: 110000			
1	2488	381	311
2	2504	453	331
3	2515	351	366
4	2525	321	319
5	2482	418	268
6	2918	312	275
7	2526	335	260
8	2649	449	285
9	2468	338	295
10	2576	400	332
Průměr	**2565,1**	**375,8**	**304,2**
σ	**127,81**	**49,49**	**31,85**
BlurSize: 100 Threshold: 120000			
1	5035	728	631
2	4935	746	539
3	4922	690	627
4	4894	621	517
5	5568	588	504
6	5046	573	495
7	4911	603	582
8	4949	741	547
9	4885	754	551
10	4895	583	528
Průměr	**5004**	**662,7**	**552,1**
σ	**195,48**	**71,9**	**45,07**

Příloha D

Tabulka č. 6 s naměřenými výsledky běhu programu pro 2 jádrový procesor.

Číslo	SerialCompute		ExecutorCompute		ForkjoinCompute	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
Molecules: 1000 Threshold: 40						
1	974	812	578	672	566	450
2	983	781	562	582	521	425
3	944	784	607	687	596	536
4	929	798	584	579	514	495
5	995	857	828	677	633	442
6	1018	793	649	490	596	548
7	978	817	627	612	509	524
8	990	806	700	557	573	528
9	996	796	620	626	592	441
10	982	805	656	522	598	418
Průměr	**978,9**	**804,9**	**641,1**	**600,4**	**569,8**	**480,7**
σ	**24,4294**	**20,47**	**73,48**	**63,48**	**39,84**	**47,91**
Molecules: 10000 Threshold: 120						
1	68193	44914	43023	28602	38991	24835
2	68430	44975	43608	28606	37460	24959
3	64315	44489	41726	28753	37427	24701
4	64043	44888	50778	28502	37877	24961
5	68210	43192	44519	28727	37168	24784
6	70101	43336	43606	28509	37130	25024
7	68442	43396	44158	28541	37425	24402
8	68375	44952	43191	28441	38935	25227
9	68267	45055	43088	28693	40880	24930
10	68091	44921	43979	28751	37352	24918
Průměr	**67646,7**	**44411,8**	**44167,6**	**28612,5**	**38064,5**	**24874,1**
σ	**1818,9295**	**737,84**	**2321,12**	**107,76**	**1137,71**	**207,2**

Tabulka č. 7 s naměřenými výsledky běhu programu pro 4 jádrový procesor.

Číslo	SerialCompute		ExecutorCompute		ForkjoinCompute	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
Molecules: 1000 Threshold: 40						
1	326	314	182	168	159	147
2	332	338	173	168	164	158
3	335	346	187	177	165	142
4	329	319	188	196	164	153
5	335	312	189	196	167	178
6	327	308	185	189	163	179
7	328	317	174	155	166	153
8	334	354	185	164	170	145
9	330	329	184	160	155	157
10	329	307	179	172	174	162
Průměr	**330,5**	**324,4**	**182,6**	**174,5**	**164,7**	**157,4**
σ	**3,14**	**15,71**	**5,31**	**13,91**	**5,02**	**12,04**
Molecules: 10000 Threshold: 120						
1	22288	17222	10744	7888	9808	7486
2	22422	18303	10797	7834	10168	7483
3	22278	18589	10769	7886	9576	7519
4	22385	18329	10623	7799	10229	6968
5	22074	17268	10633	7830	9512	7405
6	22334	16951	10746	7762	9680	6976
7	22281	18461	10879	7964	9978	7261
8	22583	18354	10774	7935	9286	7549
9	22481	17272	10809	7885	9575	7392
10	22496	17255	10768	7689	9979	7220
Průměr	**22362,2**	**17800,4**	**10754,2**	**7847,2**	**9779,1**	**7325,9**
σ	**137,08**	**617,49**	**73**	**78,12**	**290,04**	**203,46**

Tabulka č. 8 s naměřenými výsledky běhu programu pro 8 jádrový procesor.

Číslo	SerialCompute		ExecutorCompute		ForkjoinCompute	
	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]	Windows [ms]	Linux [ms]
Molecules: 1000 Threshold: 20						
1	491	410	158	218	151	173
2	406	409	160	166	148	162
3	429	433	154	184	162	154
4	393	434	165	186	173	169
5	433	452	314	190	156	165
6	438	403	205	195	151	162
7	413	458	170	173	150	161
8	447	450	147	182	164	161
9	431	427	237	215	160	173
10	400	426	167	195	154	163
Průměr	**428,1**	**430,2**	**187,7**	**190,4**	**156,9**	**164,3**
σ	**26,80**	**18,13**	**49,38**	**15,62**	**7,42**	**5,60**
Molecules: 10000 Threshold: 120						
1	22414	23027	7238	8352	6365	7332
2	22444	23050	7288	8440	6495	7083
3	22005	23275	7306	8481	6316	7614
4	22118	23207	7237	8498	6353	7341
5	22518	21641	7410	8419	6216	7052
6	22007	23039	7108	8399	6389	6948
7	21967	21914	7364	8257	6254	7111
8	22085	21531	7322	8409	6147	7159
9	21700	23176	7269	8346	6445	7441
10	21990	23243	7363	8422	6271	6933
Průměr	**22124,8**	**22710,3**	**7290,5**	**8402,3**	**6325,1**	**7201,4**
σ	**243,62**	**675,15**	**80,97**	**66,56**	**100,69**	**211,37**

Tabulka č. 9 s naměřenými výsledky běhu programu pro 16 jádrový procesor.

Číslo	SerialCompute	ExecutorCompute	ForkjoinCompute
	Linux [ms]	**Linux [ms]**	**Linux [ms]**
Molecules: 1000 Threshold: 40			
1	376	128	104
2	387	131	115
3	387	155	131
4	383	163	135
5	379	139	115
6	387	155	131
7	375	123	103
8	383	127	103
9	383	143	123
10	376	132	104
Průměr	**381,6**	**139,6**	**116,4**
σ	**4,54**	**13,17**	**12,19**
Molecules: 10000 Threshold: 115			
1	21824	3524	2800
2	20875	3851	3215
3	22503	3675	3171
4	22243	3987	3383
5	22463	3791	3243
6	22675	3731	3027
7	21587	3655	3259
8	21059	4083	3163
9	20891	3623	3159
10	20864	3712	3312
Průměr	**21698,4**	**3763,2**	**3173,2**
σ	**703,24**	**161,51**	**154,35**

Tabulka č. 10 s naměřenými výsledky běhu programu pro 24 jádrový procesor.

Číslo	SerialCompute	ExecutorCompute	ForkjoinCompute
	Linux [ms]	Linux [ms]	Linux [ms]
Molecules: 1000 Threshold: 40			
1	391	99	79
2	434	100	84
3	374	95	72
4	350	100	74
5	361	88	63
6	563	109	96
7	376	107	86
8	363	91	76
9	342	134	106
10	390	103	76
Průměr	**394,4**	**102,6**	**81,2**
σ	**61,32**	**12,16**	**11,80**
Molecules: 10000 Threshold: 110			
1	18565	2145	1641
2	19217	1965	1708
3	18087	2106	1653
4	19271	2096	1705
5	17487	2063	1604
6	17401	1832	1625
7	18998	2233	1662
8	18242	1975	1633
9	18002	2283	1779
10	17930	2159	1624
Průměr	**18320**	**2085,7**	**1663,4**
σ	**638,81**	**127,17**	**50,17**