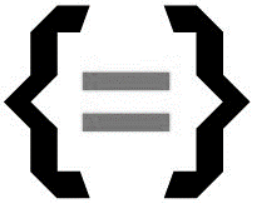


Univerzita Hradec Králové
Fakulta informatiky a managementu

Katedra informačních technologií

Využití heterogenních výpočetních systémů pro rozsáhlé agentové simulace

---

Disertační práce

Autor: Ing. Jan Procházka

Studijní program: P1802 Aplikovaná informatika

Studijní obor: 1802V001 Aplikovaná informatika

Školitelka: doc. RNDr. Kamila Štekerová, Ph.D.

Katedra/pracoviště školitele: Katedra informačních technologií

Hradec Králové 2017

Prohlášení:

Prohlašuji, že jsem tuto disertační práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 26.9.2017 ……

Anotace:

Tato disertační práce se zabývá použitím heterogenních výpočetních systémů pro rozsáhlé multi-agentové simulace. Specificky je zaměřena na paralelizaci multi-agentových modelů prostřednictvím obecných výpočtů na grafických kartách. Cílem této práce je významné urychlení simulací velkého rozsahu, pokud jde o velikost populace agentů, rozsáhlost prostředí modelu a množství simulačních kroků, které je třeba vykonat. Prvním výstupem disertačního výzkumu je vytvořený metodický postup pro návrh a implementaci paralelně pracujících modelů. Tento metodický postup je postaven na vybraných principech, nástrojích a technikách softwarového návrhu, agentového modelování a na Fosterově metodice návrhu paralelních aplikací. Druhým prezentovaným výstupem je nástroj NL2OCL, nově vytvořené programové rozšíření pro systém NetLogo, které umožňuje realizaci paralelních výpočtů na grafických kartách prostřednictvím platformy OpenCL přímo z prostředí systému NetLogo. Nakonec práce demonstruje použití metodického postupu a nástroje NL2OCL na vybraných modelech. Na základě výsledků provedených experimentů nad těmito modely práce potvrzuje hypotézu, že je možné využít prostředky heterogenních výpočetních systémů pro významné urychlení rozsáhlých multi-agentových simulací v prostředí standardních konfigurací osobních počítačů.

Klíčová slova: multi-agentové modelování, rozsáhlé simulace, heterogenní výpočetní systémy, paralelní aplikace, GPGPU, OpenCL, NetLogo

Annotation:

This dissertation thesis deals with heterogenous computing systems in large-scale multi-agent simulations. Specifically, it focuses on model parallelization via general purpose computations on graphical processor units. The main aim is to provide a significant speedup to large simulations with huge agent populations, large model environments and high number of required simulation steps. The first research output presented is a new methodology for designing and implementing parallelized multi-agent models. It combines selected software design and agent-based modelling approaches, tools, and techniques with Foster's parallel application design methodology. The second output is NL2OCL – a newly developed NetLogo extension that allows NetLogo models to execute parallel computations on GPU using OpenCL platform directly from NetLogo system. Finally, the thesis demonstrates the new methodology and NL2OCL software tool on selected models. Based on experimental results achieved with these models the thesis confirms the hypothesis that it is possible to use heterogenous computational systems to significantly increase simulation speed of large multi-agent models on standard PC configurations.

Keywords: multi-agent modelling, large-scale simulations, heterogenous computing systems, parallel applications, GPGPU, OpenCL, NetLogo

Poděkování:

Děkuji školitelce doc. RNDr. Kamile Štekerové, Ph.D. za všestranné vedení po celou dobu mého doktorského studia. Velmi si vážím odborných rad, které mi poskytla při směrování a realizaci mého disertačního výzkumu. Děkuji jí ale především za velkou kolegialitu a trpělivost, jejíž hranice jsem se občas, jako její školenec, snažil svým počínáním narušit (vždy neúspěšně).

Tato disertační práce by nevznikla bez obrovského přispění mé rodiny. Podpora mé manželky a dcery pro mě představovala rozhodující energii ve chvílích, kdy se zdálo, že dál už to nepůjde. Díky nim se vždy nějaká cesta dál objevila. „Díky, světýlka moje!“

Rád bych vyjádřil velké díky také svým rodičům za jejich celoživotní podporu všech mých aktivit spojených se studiem. To oni mi chuť zkoumat a objevovat předali a já se budu snažit předávat tuto chuť dál.

Obsah

1 Úvod

Tato disertační práce se zabývá teoretickými a praktickými aspekty paralelního zpracování výpočtů v multi-agentových simulacích. Její součástí je představení vytvořeného metodického postupu pro návrh a implementaci multi-agentových modelů s paralelními výpočty. Dále práce prezentuje ucelené softwarového řešení v podobě nástroje NL2OCL, programového rozšíření pro systém NetLogo, založeného na platformě OpenCL, které modelům v NetLogu paralelní výpočty umožnuje realizovat. Práce demonstruje použití metodického postupu a nástroje NL2OCL, na ukázkách paralelizace vybraných multi-agentových modelů.

Text je rozdělen do osmi hlavních kapitol:

- Úvod popisuje strukturu práce a vysvětluje motivaci autora pro volbu daného disertačního tématu v kontextu jeho účasti na konkrétních výzkumných projektech.
- Po úvodu je v kapitole 2 představen hlavní cíl disertační práce spolu s třemi dílčími cíli. Dále je zde popsána metodika, která byla pro disertační výzkum použita.
- V kapitole 3 následuje analýza současného stavu výzkumu v oblastech týkajících se disertačního tématu. Jde o tyto oblasti:
    - *Hardwarové předpoklady paralelizace* – kapitola 3.1 nastíní cestu zvyšování výkonu výpočetních prostředků a proces nahrazování jedno-jádrových výpočetních jednotek více – jádrovými.
    - *Datové struktury a synchronizace* – kapitola 3.2 se věnuje přechodu od sekvenčního ke konkurenčnímu zpracování výpočtů a problematice sdílených datových struktur.
    - *Paralelní aplikace* – kapitola 3.3 se zabývá teoretickými aspekty paralelních aplikací, jejich taxonomií, výkonnostními metrikami a zákony popisujícími zrychlování výpočtů pomocí paralelizace.
    - *Fosterova metodika návrhu paralelních aplikací* je rozebrána v kapitole 3.4 jako jeden ze stavebních kamenů metodického postupu pro návrh paralelně pracujících multi-agentových systémů, který je jedním z výstupů této disertační práce.
    - Kapitola 3.5 pojednává o *heterogenních výpočetních systémech* a o využití grafických karet pro paralelní výpočty. Zvláštní pozornost je věnována platformě OpenCL, která byla vybrána jako implementační platforma k vytvoření softwarového nástroje NL2OCL.
    - *Multi-agentové modely a simulace* – kapitola 3.6 pojednává o principech návrhu multi-agentových modelů a aspektech týkajících se jejich paralelizace.
- Kapitola 4 představí vytvořený metodický postup pro návrh a implementaci multi-agentových systémů s paralelními výpočty.

- V kapitole 5 následuje prezentace softwarového nástroje NL2OCL, který umožňuje paralelizaci výpočtů v multi-agentových modelech pomocí heterogenních výpočtů.
- Kapitola 6 je věnována demonstraci použití metodického postupu a programového nástroje NL2OCL při paralelizaci vybraných modelů:
    - Model *HEJNA* (kap. 6.1) – paralelizace Reynoldsova modelu hejna. Na tomto modelu je demonstrováno základní použití metodického postupu pro návrh paralelně pracujících multi-agentových modelů a programového rozšíření NL2OCL pro urychlení simulace.
    - Model *EVAKUACE* (kap. 6.2) – vytvoření nového evakuačního modelu na bázi buněčného automatu s hexagonální sítí buněk a jeho paralelizace. Na modelu je předvedeno použití metodického postupu a programového rozšíření NL2OCL pro komplexnější modely.
    - Model *OSÍDLENÍ* (kap. 6.3) – analýza možností paralelizace existujícího modelu budování sítě Keltských sídlišť. Na tomto modelu je demonstrován návrh paralelizace modelu výpočetní povahy používajícího vstupní GIS data a grafové algoritmy.
- Závěr práci shrnuje a uzavírá.

Motivace k volbě disertačního tématu souvisí se zaměřením autora na multi-agentové modelování systémů vykazujících emergentní chování a na problematiku výpočetního výkonu v těchto modelech. Během doktorského studia měl autor možnost spolupracovat na vývoji takovýchto modelů v různých oblastech. Jednalo se o dva směry výzkumu. Prvním směrem byl výzkum dynamiky davu a agentových modelů pohybu chodců, s aplikací v modelech udržitelnosti cestovního ruchu. Druhý směr se týkal použití agentových modelů v archeologii, konkrétně při studiu rozvoje sítě keltských sídlišť na českém území na konci doby železné. Při obou těchto výzkumech se opakovala situace, kdy zvyšující se komplexita a rozsah modelů neúnosně prodlužovaly běh simulací. Odtud pochází motivace pro hledání možností zrychlení výpočtů a myšlenka zabývat se paralelními výpočty v heterogenních výpočetních systémech. Přes dostupnost platforem pro paralelní výpočty mohou být vstupní bariéry pro vybudování výpočetního systému s výkonem schopným vyhovět požadavkům na rychlé zpracování velkých simulací vysoké. Využití výpočtů na výkonných grafických kartách v konfiguraci standardních PC bylo zvoleno jako vhodná alternativa pro výzkumné týmy, které nemají přístup k systémům vysokého výpočetního výkonu HPC (high performance computing).

2 Cíle a metodika disertačního výzkumu

Hlavním cílem této disertační práce je:

Umožnit využití prostředků heterogenních výpočetních systémů pro významné urychlení rozsáhlých multi-agentových simulací v prostředí standardních konfigurací osobních počítačů.

Specificky pro tento cíl platí:

- Prostředky heterogenních výpočetních systémů je v této práci myšleno zejména propojení multi-agentového systému s paralelními výpočty na grafických kartách.
- Za rozsáhlé multi-agentové simulace jsou považovány simulace s populacemi od 10 000 do 100 000 agentů.
- Významným urychlením simulace je pro takto velké populace agentů myšleno urychlení o více než 100 % (tj. alespoň dvojnásobná rychlost simulace).
- Standardními konfiguracemi osobních počítačů jsou myšleny výpočetní systémy, které nelze označit jako systémy vysokého výkonu. Jediným nadstandardním prvkem těchto konfigurací je výkonná grafická karta.

Dosažení hlavního cíle disertační práce je umožněno prostřednictvím splnění těchto tří dílčích cílů:

1. **Navrhnout metodický postup pro návrh, implementaci a testování multi-agentových simulací velkého rozsahu s paralelními výpočty v heterogenních výpočetních systémech (s výpočty na grafických kartách).**

   Tento metodický postup je hlavním teoretickým výsledkem disertační práce. Vychází z Fosterovy metodiky návrhu paralelních aplikací a odráží praktické zkušenosti autora s vytvářením konkrétních paralelizovaných multi-agentových modelů.

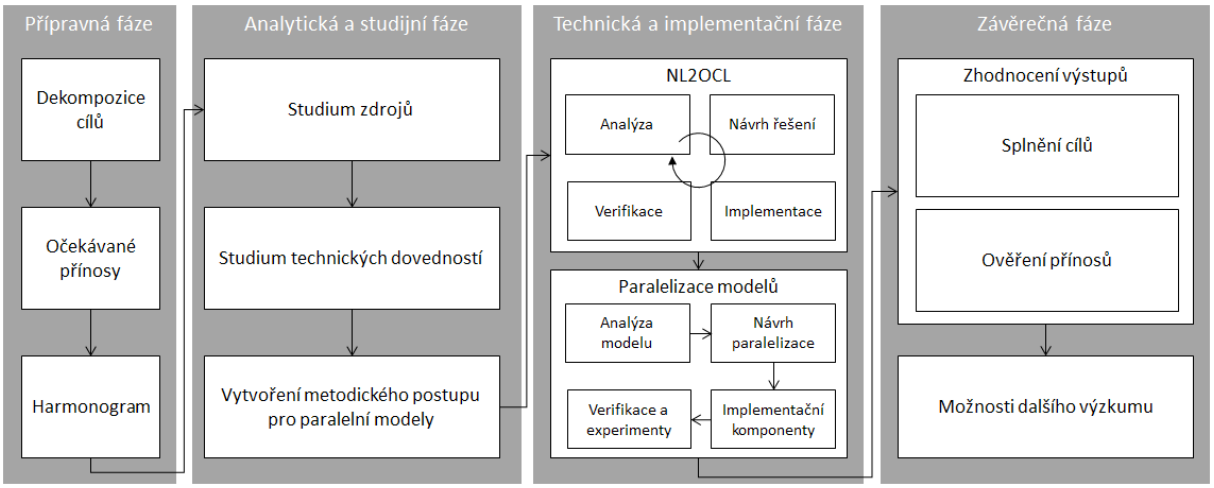
2. **Vytvořit programové rozšíření pro implementaci paralelních výpočtů na grafických kartách v prostředí multi-agentových simulací.**

   Vytvoření softwarové řešení pro realizaci paralelních výpočtů v multi-agentových simulacích v návaznosti na metodický postup formou programového rozšíření NL2OCL pro systém NetLogo. Využití otevřené platformy pro heterogenní výpočetní systémy OpenCL.

3. **Ověřit použití metodického postupu a programového rozšíření NL2OCL na konkrétních modelech.**

   Transformace vybraných modelů na verze s paralelizovanými výpočty a provedení simulací většího rozsahu nad těmito modely.

Disertační výzkum, jehož výsledky jsou v této práci prezentovány, probíhal podle metodiky, která je schematicky zobrazena na obr. 1. Tato metodika je rozdělena do čtyř vzájemně propojených fází: přípravné, analyticko-studijní, technicko-implementační a závěrečné.



Obrázek 1 – Metodika disertačního výzkumu, zdroj: Autor

Přípravná fáze:

- Dekompozice cílů disertační práce – rozdělení cílů na dílčí uchopitelné celky.
- Vyjasnění očekávaných přínosů pro jednotlivé dílčí cíle:
    - V rámci vytvoření metodického postupu pro návrh a implementaci paralelních multi-agentových modelů dojde ke konkretizaci použití jednotlivých kroků obecné Fosterovy metodiky pro případ návrhu agentových modelů. Bude poskytnut ucelený pohledu na vývojový cyklus agentových modelů s paralelními výpočty.
    - Díky vývoji programového rozšíření NL2OCL budou výkonnostní benefity paralelních výpočtů na grafických kartách zpřístupněny modelům v simulačních systémech postavených na jazyku Java. Programové rozšíření bude naplňovat potřeby jednotlivých kroků metodického postupu návrhu a vývoje multi-agentových systémů s paralelními výpočty. Proběhne rozšíření povědomí o existenci tohoto nástroje tak, aby mohli přispět i další autoři. Budou navázány vztahy s vývojovými týmy i jednotlivci zabývajícími se paralelizací výpočtů v agentových simulacích.
    - Dojde k paralelizace vybraných modelů. Tím bude prověřena použitelnost navrženého metodického postupu a technického řešení programového rozšíření NL2OLC při paralelizaci konkrétních modelů. Dojde k ohodnocení aspektu udržitelnosti nákladů na vývoj a úpravu modelů a k ověření možností automatizace některých implementačních kroků. Hypotéza, že se simulace velkého rozsahu dají provádět i v prostředí standardních PC za pomoci výpočtů na výkonných grafických kartách, bude potvrzena

nebo vyvrácena. Dojde k ověření reálných možností využití trendu vývoje grafických karet pro vědeckou práci.

Analyticko-studijní fáze:

- Budou shromážděny a analyzovány dostupné práce a literatura mapující současný stav pro relevantních oblasti výzkumu:
    - Přístupy k multi-agentovému modelování.
    - Možnosti paralelizace aplikací.
    - Programování více a mnoho-jádrových výpočetních systémů.
    - Obecné výpočty na grafických kartách.
    - Platformy pro heterogenní výpočty, OpenCL platforma.
- Proběhne nastudování následujících technických dovedností potřebných pro realizaci programového rozšíření NL2OCL:
    - Vytváření programových rozšíření pro NetLogo (NetLogo API).
    - Propojení technologie NetLogo – Java – C++ – OpenCL (JNI API, OpenCL API).
    - Programování OpenCL kernelů (programovací jazyk C-OpenCL).
- Dojde k vytvoření prvního dílčího disertačního cíle, tj. metodického postupu pro vytváření paralelně pracujících multi-agentových modelů.

Technicko-implementační fáze:

- Programové rozšíření NL2OCL bude vytvořeno v několika vývojových cyklech, skládajících se z kroků: Analýza požadavků -> Návrh řešení -> Implementace -> Verifikace.
- Proběhne paralelizace vybraných modelů. Nejprve budou modely analyzovány z pohledu návrhu, poté budou identifikovány možnosti paralelizace na základě vodítek z metodického postupu. Budou připraveny jednotlivé implementační části pro výsledný paralelně pracující model (aktualizace modelu v NetLogu, příprava OpenCL kernelů).
- Dojde k otestování funkčnosti modelu a budou navrženy a poté realizovány experimenty.

Závěrečná fáze:

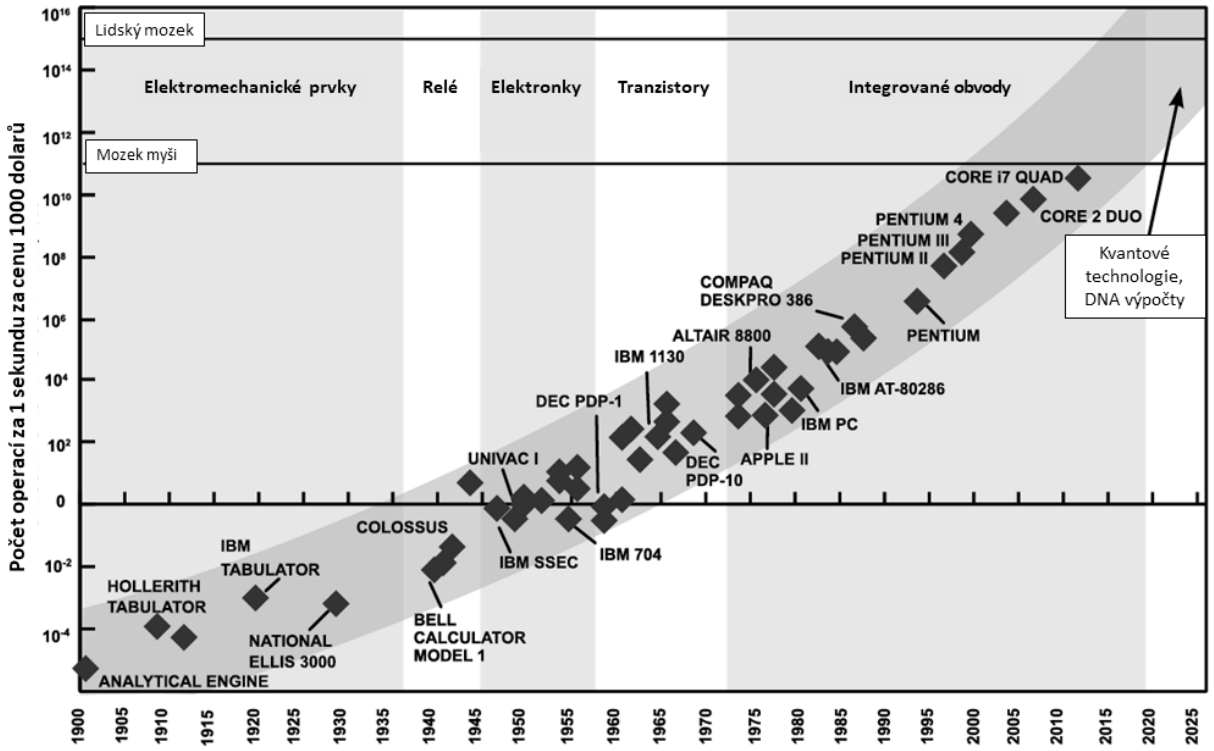
- Dosažené výsledky budou podrobeny diskuzi a proběhne hodnocení splnění cílů. Zároveň dojde k ověření dosažení očekávaných přínosů.
- Budou nastíněny možnosti dalšího výzkumu.

3 Analýza současného stavu

3.1 Hardwarové předpoklady paralelizace

3.1.1 Moorův zákon a Koomeyův zákon

Moorův zákon říká, že „počet tranzistorů umístěných na integrovaný obvod se při zachování stejné ceny zhruba každých 18 měsíců zdvojnásobí“ (Moor, 1965). Moor původně omezil platnost své předpovědi na 10 let, naplňuje se však dodnes, jak je patrné z obr. 1.



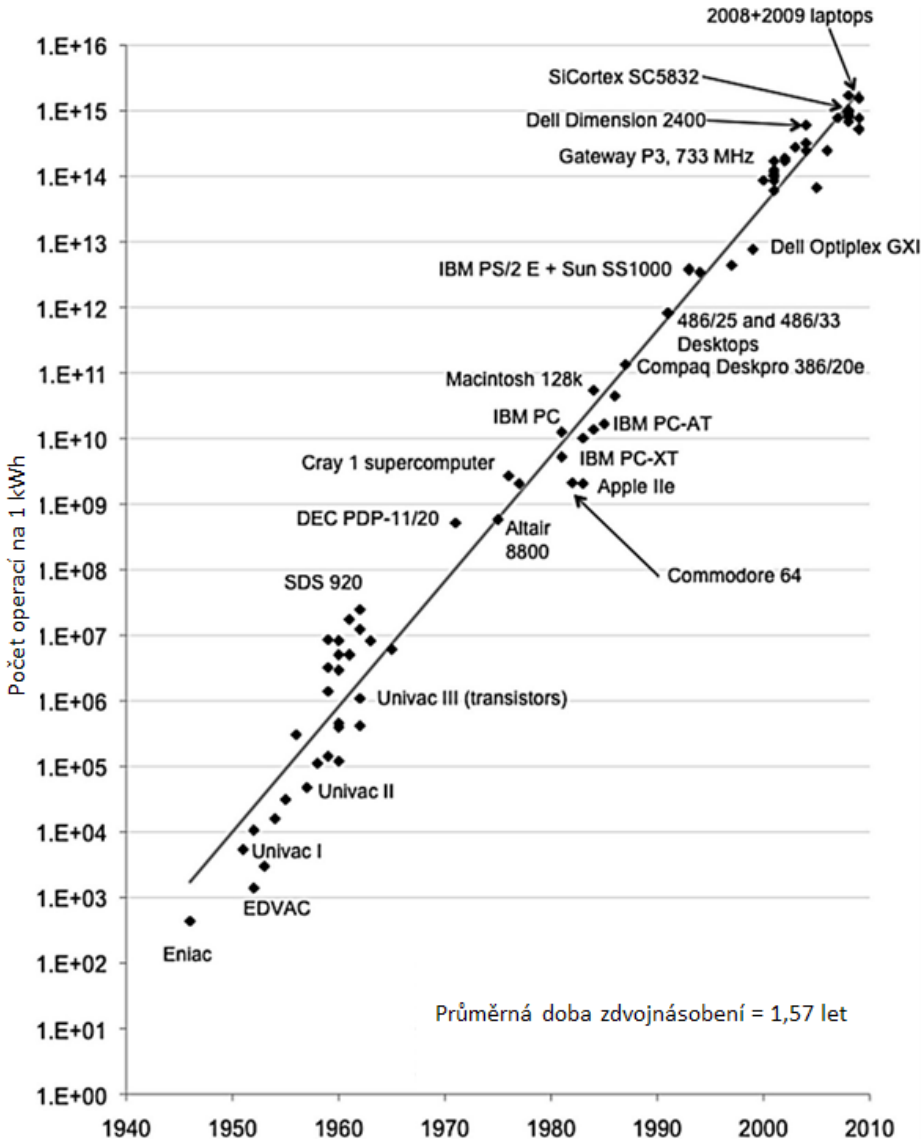
Obrázek 2 - Moorův zákon, zdroj: upraveno podle (Berezin, 2013)

Po desítky let se inovace v oblasti polovodičů odehrávaly ve třech kategoriích – menší, rychlejší a levnější. Tento vývoj odpovídal Moorovu zákonu. Velkou metou byl souboj mezi Intelem a AMD o první procesor s frekvencí vyšší než **1 GHz**. Tento souboj vyhrálo AMD v roce 2000 se svým čipem Athlon. V současné době již frekvence nejrychlejších procesorů překonává hranici 8 GHz. Ve statistikách rekordů (CPU-Z, 2016) je záznam o aktuálním vítězi z roku 2012 – jde o procesor AMD FX-8350 s frekvencí 8,79Ghz. Nejrychlejší procesor Intelu figuruje na 3. místě, záznam je z roku 2013 a jde o procesor Celeron 352 s frekvencí 8,54 GHz. Podle statistik se tedy zdá, že rychlostní závod pomalu končí, už 5 let se vítěz nemění.

Skutečná budoucnost polovodičů, alespoň podle aktuálních předpovědí Intelu, spočívá v omezení absolutních výkonů výměnou za snížení spotřeby, jak je nastíněno v článku (Merrit, 2016). Začíná nový závod, který nese označení **0 W**. Rychlost a spotřeba jsou silně vázány na použitou technologii výroby.

Existuje široké spektrum post-CMOS technologií, které dokáží snížit spotřebu, ale daní je bohužel jejich nižší rychlost. Hlavním argumentem pro použití těchto technologií je fakt, že rychlostní ztráty mohou být vykompenzovány při dostatečně rozsáhlém nasazení těchto technologií. Tento pohled se stává čím dál tím relevantnějším, zejména při snahách o návrh exascale výpočetních systémů (tj. systémů s výpočetním výkonem alespoň jeden exaFLOPS, 10^{18} FLOPS). Příliš velká spotřeba energie v poměru k hustotě integrace je hlavním důvodem, proč je dnes prakticky nereálné zkonstruovat hardware, který by měl dostatečnou hustotu integrace k vybudování exascale výpočetního systému.

Vedle Moroova zákona existuje zákon jiný, který s ním úzce souvisí a sdílí s ním stejné, exponenciální tempo růstu – jde o Koomeyův zákon. Jonathan Koomey popsal v roce 2011 trend, který můžeme již 60 let pozorovat: „Elektrická efektivita výpočetních prostředků se zdvojnásobuje přibližně každých 18 měsíců.“ (Koomey at al., 2009).

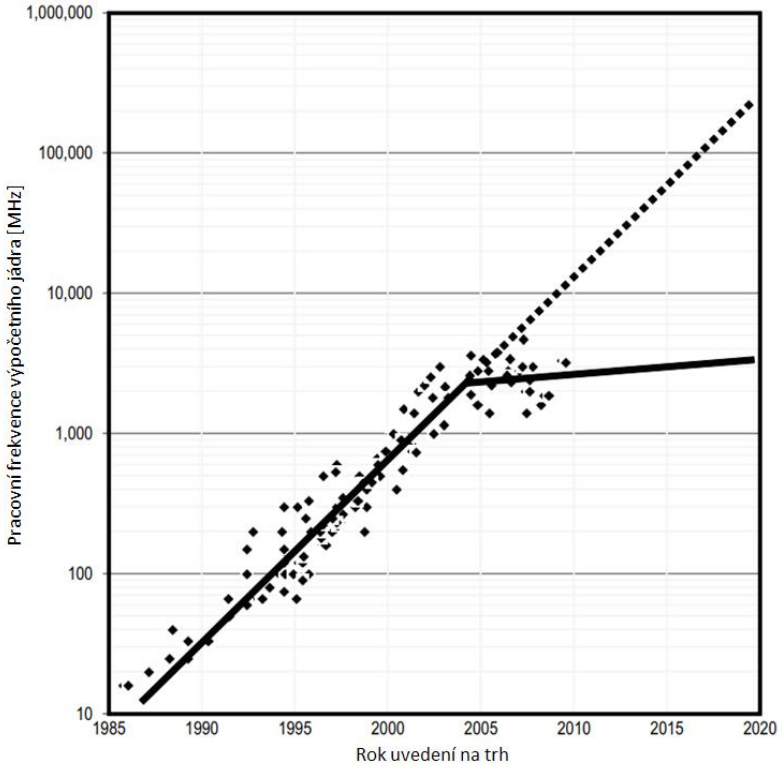


Obrázek 3 – Vývoj počtu operací při spotřebě 1 kWh, zdroj: upraveno podle (Koomey at al., 2009)

3.1.2 Rozvoj více-jádrových výpočetních jednotek

Organizace ITRS (International Technology Roadmap for Semiconductors), za kterou stojí vůdčí světoví výrobci polovodičové techniky, je autorem zprávy, předpovídající v rozmezí let 2010-2020 pokračující trend růstu počtu tranzistorů v integrovaných obvodech (ITRS, 2009). Tato zpráva ale současně predikuje, že tento růst nebude uskutečněno pouze na základě zvyšující se hustoty integrace (počtu tranzistorů na jednotku plochy), tak jak tomu bylo přibližně do roku 2005. Na obr. 3 je zachycen historický vývoj výkonu (pracovní frekvence) jedno-jádrových procesorů a predikce do roku 2020. Závěry zprávy se dají shrnout do dvou hlavních zjištění, které lze z pohledu současného vývoje považovat za zcela potvrzené:

- Po mnoha desítkách let exponenciálního růstu výkonu jedno-jádrových procesorů je toto tempo neudržitelné a bude se snižovat.
- Rozvoj výkonu výpočetních systémů (a to i pro více-jádrové paralelní systémy) bude v následující dekádě limitován hlavně spotřebou energie.



Obrázek 4 - Pracovní frekvence jedno-jádrových výpočetních jednotek, zdroj: upraveno podle (Fuller & Millett, 2011)

Hardwarové prostředky disponující mnoha výpočetními jádry jdou dnes běžnou součástí výpočetních systémů vysokého výkonu (high-performance computing). Optimalizace těchto prostředků směřují nejen k vyššímu výkonu, ale také k úspoře energie. Rozvoj více a mnoha-jádrových výpočetní systémů

pomáhá překonávat problémy typické pro jedno-jádrové procesory. Tyto problémy jsou formulovány jako tzv. bariéry zvyšování výkonu (Fuller & Millett, 2011; Hennessy & Patterson, 2011):

- **Příkonová bariéra** – vysoké energetické nároky jedno-jádrových výpočetních jednotek kladou zvláštní požadavky na jejich zapouzdření a chlazení. Zmenšování velikosti tranzistorových hradel na integrovaných obvodech spolu se snižujícím se napájecím napětím byla úspěšná strategie vedoucí k mnohonásobnému zvýšení výpočetního výkonu jedno-jádrových výpočetních jednotek. Kolem roku 2004 ale dosáhla úroveň integrace takové míry, že pro zachování funkčnosti hradel nebylo již dále možné snižovat napájecí napětí. Další zvyšování výkonu prostřednictvím zvyšování pracovní frekvence nebylo možné bez zvyšování úrovně ztrátového tepla. Tato úroveň brzy dosáhla kritické, prakticky nepřekročitelné hranice, která byla označena za příkonovou bariéru.
- **Paměťová bariéra** – rychlost výpočetních jednotek rostla mnohem vyšším tempem, než jakým rostla rychlost pamětí. Rozdíl rychlostí dosáhl míry, která nedovolí zvyšovat rychlost celkového výpočetního systému přes zvyšující se rychlost výpočetních jednotek. Tento trend byl označen za paměťovou bariéru. Negativní efekt se sice daří řešit pomocí různých technik (např. pipelining, zpracování mimo pořadí nebo víceúrovňové vyrovnávací paměti), ale všechna tato řešení vedou zároveň ke zvyšující se komplexitě hierarchie paměti. Dnes jsou běžné tři úrovně vyrovnávací paměti pro výpočetní jednotky k potlačení efektů latence rychlostí výpočetní jednotky a operační paměti.
- **Bariéra paralelizace na úrovni instrukcí** – existují možnosti, jak paralelizovat sériový tok instrukcí zpracovávaný výpočetní jednotkou pomocí tzv. instrukčních proudů. Technologická implementace takových řešení je ale velmi náročná, častěji je tato bariéra překonávána paralelizací pomocí více výpočetních jader.
- **Bariéra komplexity** – návrh komplexních výpočetních jednotek, které disponují prostředky paralelizace na úrovni instrukcí, jejich výroba a testování jsou velmi technicky a ekonomicky náročné. Tuto bariéru překonává více-jádrový přístup tím, že návrh jednoho jádra je proveden pouze jednou a je opakovaně použit pro více-jádrové řešení. Návrh jednotlivých jader více-jádrových výpočetních jednotek je mnohem jednodušší, návrh komplexních jedno-jádrových výpočetních jednotek.

Na druhou stranu se s mnoha-jádrovými systémy objevují bariéry nové, pro které se v současné době stále hledají efektivní řešení (Pllana at al., 2017; Herlihy & Shavit, 2008). Jde především o **bariéru programovatelnosti**. Programování mnoha-jádrových výpočetních jednotek, především v heterogenních konfiguracích (více druhů vzájemně spolupracujících výpočetních jednotek), je mnohem komplexnější úkol, než je programování jedno-jádrových procesorů.

3.2 Datové struktury a synchronizace

Počítačové programy jsou tvořeny algoritmy a daty. Tento fakt tvoří název Wirthovy knihy „Algorithms + Data Structures = Programs“ (Wirth, 1976). Vztah mezi algoritmy a datovými strukturami platí dnes stejně jako před 40 lety. Zvláště potom nabývá na důležitosti pro paralelní programování a paralelní aplikace. Datové struktury s konkurenčním přístupem jsou podstatnou složkou paralelního programování. Propojení algoritmů a datových struktur se stává kritičtější při návrhu konkurenčních, paralelních programů, které pracující s mnoha výpočetními jednotkami. Algoritmy pracující s datovými strukturami jsou závislé na operacích, které je možné nad datovými strukturami provádět. To, jaké datové struktury jsou použity, určuje způsob, jakým bude k datům přistupováno (čtení a zápis) a jak budou algoritmy tato data zpracovávat. V sekvenčních programech je použití správných datových struktur rozhodujícím výkonnostním faktorem. V paralelních programech důležitost použití efektivních datových struktur ještě roste, protože uchovávaná data musejí být bezpečně sdílena mezi různými částmi paralelně běžícího výpočtu (Cederman, 2017).

3.2.1 Datové struktury se zamykáním zdrojů

Model sdílené paměti vyžaduje, aby byla zajištěna integrita dat, ke kterým přistupují a jež modifikují různá vlákna nebo procesy (Nikolakopoulos at al., 2015). V mnoha zdrojích, které se konkurenčními datovými strukturami zabývají, je popsána celá škála vhodných postupů. Přes ty jednoduché, využívající principy zamykání celé datové struktury – objektu (coarse-grained lock) přes pokročilejší techniky jemného zamykání (fine-grained lock), které datové struktury zamykají částečně za cenu náročnější implementace, až po datové struktury využívající neblokující přístupy, či kombinace předešlých přístupů (Cederman at. al, 2013; Fatourou & Kallimanis, 2012; Herlihy & Shavit, 2008). V této kapitole jsou popsány synchronizační problémy, ke kterým může dojít při použití přístupu zamykání zdrojů. Posledním jmenovaným, neblokujícím přístupům, je věnována příští kapitola.

V mnoha-vláknových systémech dochází při synchronizaci zamykáním pomocí tzv. mutexů (mutual exclusion) k problémům, které nastávají v situacích, do který se takový systém může díky uzamčením dostat. Problémy uzamčení se dají zobecnit do vzorových situací (Herlihy & Shavit, 2008; Pirkelbauer at al., 2016; Cederman at al., 2017), společnou charakteristikou těchto situací je to, že výpočet nepokračuje směrem k dokončení očekávaným způsobem:

- **Uváznutí** (deadlock) – k uváznutí dojde, pokud jsou dvě nebo více vláken zablokovány vzájemným čekáním na uvolnění zdroje, který používá jiné vlákno této skupiny. Situace je zobrazena na obr. 5(a). Vlákno T1 má přidělen zdroj R1 a vlákno Tě má přidělen zdroj R2. Vlákno T1 se snaží získat zdroj R2, ale musí čekat na uvolnění od T2, současně T2 se pokouší získat zdroj R1, ale musí čekat na jeho uvolnění od T1. Vlákna jsou navzájem zablokována

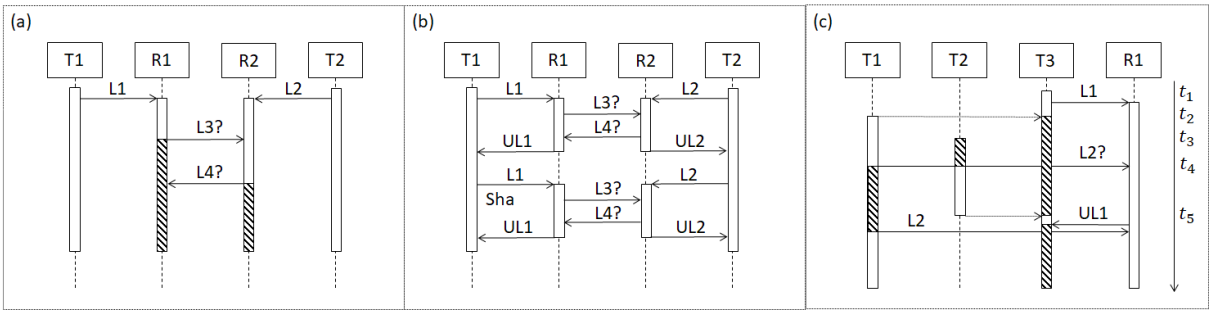
čekáním na své zdroje, které ale nikdy neuvolní – výpočet uváznul. Vnik zablokování je vázán na splnění tzv. Coffmanových podmínek (Coffman at al., 1971):

- vzájemné vyloučení – prostředek používá jen jedno vlákno nebo žádné,
- držení zdrojů a čekání – vlákna vlastnící prostředky mohou žádat o další,
- neodnímatelnost – přidělené prostředky nemohou být odebrány,
- kruhová závislost – existuje kruhový řetěz vláken, ve kterém každé vlákno čeká na prostředek vlastněný jiným vláknem.

Možné způsoby prevence uváznutí vycházejí z implementace mechanizmu, který porušení alespoň jednu z Coffmanových podmínek.

- **Živé uváznutí** (livelock) – tato situace je zachycena na obr. 5(b) - vlákno T1 vlastní zdroj R1 a pokusí získat zdroj R2, který je ale přiřazen druhému vláknu T2. T1 reaguje tak, že uvolní i zdroj R1. Než ale T1 svůj zdroj R1 uvolní, pokusí se zdroj R1 získat druhé vlákno T2 ke svému již vlastněnému zdroji R2. Protože je v tu dobu zdroj R1 nedostupný, T2 uvolní svůj již vlastněný zdroj R2. Obě vlána jsou tedy bez zdrojů. Celý cyklus se potom opakuje. Nikdy nedojde k přiřazení obou zdrojů jednomu z vláken, aby mohlo dokončit požadovaný výpočet. Obě vlákna sice pracují, ale fakticky se výpočet nikam neposouvá. Detekce livelocku je komplikovanější, protože vlákna při něm na rozdíl od deadlocku pracují (analogií může být situace, kdy si dva lidé dávají ve dveřích vzájemně přednost, aniž by skutečně dveřmi někdo z nich prošel).
- **Záměna priorit** – k této situaci dojde, pokud při konkurenčním zpracování dojde k tomu, že místo vlákna s vyšší prioritou běží vlákno s prioritou nižší. Na obr. 5(c) je tato situace zachycena. Priority vláken jsou $\mathrm{P(T1)} > \mathrm{P(T2)} > \mathrm{P(T3)}$. Nejprve se v čase t_1 rozběhne vlákno T3 s nejnižší prioritou, neběží žádná jiná vlákna s vyšší prioritou, proto může T3 běžet. T3 získá na začátku své činnosti zdroj R1 a uzamkne ho (L1). V čase t_2 se rozběhne vlákno T1, které má nejvyšší prioritu, proto se přeruší průběh vlákna T3. V čase t_3 se vytvoří vlákno T2 se střední prioritou, ale neběží, protože běží vlákno T1, které má vyšší prioritu. V čase t_4 vlákno T1 potřebuje ke svému běhu zdroj R1, který se snaží získat (L2?). Vlákno T3 ale zdroj R1 neuvolní, místo toho, aby se vlákno T3 rozběhlo a R1 uvolnilo, poběží místo něj vlákno T2, protože má vyšší prioritu než T1. Zde právě dochází k záměně priorit. Místo vlákna s nejvyšší prioritou T1 běží vlákno se střední prioritou T2. Vlákno T2 ukončí v čase t_5 svůj běh a umožní vláknu T1 pokračovat a uvolnit zdroj R1 pro vlákno T1, poté je vlákno T3 opět pozastaveno a běží (již správně) vlákno T1 s nejvyšší prioritou. Řešení tohoto problematického scénáře je možné metodou „dědění priority“, která je popsána v (Sha et al., 1990). Problém záměny priorit je méně častý a hůře detekovatelný než první dva synchronizační problémy. Známým případem

výskytu tohoto problému z roku 1997 byly potíže operačního systému VxWorks ovládajícího sondu Mars Pathfinder. Během testování se problém nepodařilo odhalit, sonda již byla na povrchu Marsu, když k problematickému scénáři došlo. Hledání chyby a oprava probíhaly na dálku ze Země (Durkin, 1998).



Obrázek 5 - Problematické scénáře použití uzamčení: (a) uváznutí, (b) živé uváznutí, (c) záměna priorit, zdroj: Autor

3.2.2 Neblokující datové struktury

V poslední dekádě bylo publikováno mnoho prací zabývajících se studiem a návrhem algoritmů a datových struktur využívajících sdílené datové struktury a sdílenou paměť, které nepoužívají principy uzamykání zdrojů. Motivace k hledání nových datových struktur vyhází ze specifických potřeb paralelních algoritmů a paralelního zpracování dat. Jejich konstrukce vychází z požadavků kladených na nové aplikace, aby se jejich algoritmy změnily a přizpůsobily vývoji a změnám hardwarových architektur (Laborde et al., 2017).

Vývojáři, kteří se konkurenčně zpracovávaným kódem zabývají, jsou postaveni před problémy, které nemuseli při sekvenčním programování řešit. Nejdůležitějším z těchto problémů je správná manipulace se sdílenými daty (Laborde et al., 2017). Aktuálně je oblast výzkumu nových datových struktur pro více-jádrové a mnoho-jádrové výpočetní systémy velice živá. Publikace vycházející z tohoto výzkumu poskytují detailní návrhy datových struktur, které se běžně používají při sekvenčním zpracování, v jejich neblokující podobě, jež je vhodná pro paralelní zpracování - např. zásobník (Goel, 2016), fronta (Pirkelbauer at al., 2016; Zhang & Dechev, 2016) nebo hashovací mapa (Laborde et al., 2017). Doposud používané techniky synchronizace datových struktur pomocí zamykání představují pro paralelní aplikace velkou výkonnostní překážku. Přínosem nových datových struktur je překonávání těchto překážek a posun algoritmů od těch, které pracují s nesynchronizovanými datovými strukturami, nebo s datovými strukturami se zamykáním, k neblokujícím přístupům. Implementace takovýchto neblokujících datových struktur je však komplikovanější než jejich sekvenčně pracující předobrazy. Neblokující přístupy se snaží problémům se zamykáním zdrojů předcházet, to znamená, že při jejich použití k problémovým situacím popsaným v předešlé kapitole nemůže vůbec dojít.

Datové struktury a algoritmy, které nad nimi probíhají se z pohledu blokování či neblokování mohou rozdělit do několika skupin a podskupin podle způsobu, jakým jsou naplněny vlastnosti pokračování či nepokračování výpočetních funkcí, tzv. progress conditions (Herlihy a Shavit 2008). Tyto vlastnosti se dělí na závislé a nezávislé, podle toho, zda vyšetřovaná vlastnost funkce existuje sama o sobě nebo je výsledkem spolupůsobení jiných funkcí a vláken. Toto jsou jednotlivé vlastnosti:

- **Vlastnost blokování** (blocking) – závislá vlastnost. Datové struktury a algoritmy používající některý z mechanizmů uzamykání přidělených zdrojů (kritické sekce, semafory) jsou blokující. Problémy, ke kterým může dojít byly představeny v předchozí kapitole – zablokování, živé zablokování, inverze priorit).

  Definice: Funkce je označena za blokační, pokud může nastat situace, ve které vlákno, které tuto funkci vykonává, nemůže pokračovat ve svém běhu, dokud některé z jiných vláken neuvolní požadovaný zdroj.
- **Vlastnost neumořování** (starvation-free) – závislá vlastnost. Splnění této podmínky je závislé přímo na platformě a operačním systému, na kterých je funkce vykonávána. Platforma a/nebo operační systém musí garantovat, že bude tato podmínka zabezpečena (např. pomocí vhodných atomických instrukcí).

  Definice: Pokud se některé vlákno nachází v kritické sekci (vlastní zámek k nějakému zdroji), potom některé jiné vlákno, které se pokouší do této kritické sekce dostat (využít zamčený zdroj), po konečném počtu výpočetních kroků uspěje, a to bez ohledu na to, zda vlákno nacházející se v kritické sekci doběhlo nebo ne (může být trvale pozastaveno).
- **Vlastnost bezpřekážkovosti** (obstruction-free) – nezávislá vlastnost. Tato vlastnost nezávisí na koexistenci vlákna vykonávajícího funkci s jinými vlákny. Je to izolovaná, nezávislá vlastnost jednoho vlákna vykonávající jednu funkci.

  Definice: Funkce je bezpřekážková, pokud je při izolovaném spuštění vždy dokončena v konečném počtu výpočetních kroků.
- **Vlastnost nezablokování** (lock-free) – nezávislá vlastnost. Splnění této vlastnosti garantuje, že výpočet bude vždy pokračovat kupředu směrem k dokončení. To znamená, že alespoň jedno vlákno bude vždy vykonávat kroky, které jsou pro výpočet nějakým přínosem, tj. nebude konat jen kroky vedoucí výhradně k umoření zdrojů, jak tomu je např. při živém uváznutí (při live-locku).

  Definice: Funkce má lock-free vlastnost, pokud pro nekonečný počet volání této funkce některé vlákno, které funkci volá dokončí svoji činnost v konečném počtu kroků.
- **Vlastnost nečekání** (wait-free) – nezávislá vlastnost. Splnění této vlastnosti garantuje, že všechna vlákna volající funkci během přidělovaných výpočetních časů vykonávají určitý progres

výpočtu a výpočet nakonec v konečném celkovém čase dokončí. Tato vlastnost zaručuje, že celkový počet kroků potřebný pro dokončení všech vláken bude konečný. Tento konečný počet, závisející hlavně na počtu funkci volajících vláken, může být velmi velký.

Definice: Funkce je wait-free, pokud je dokončena všemi vlákny, které ji volají v konečném počtu kroků.

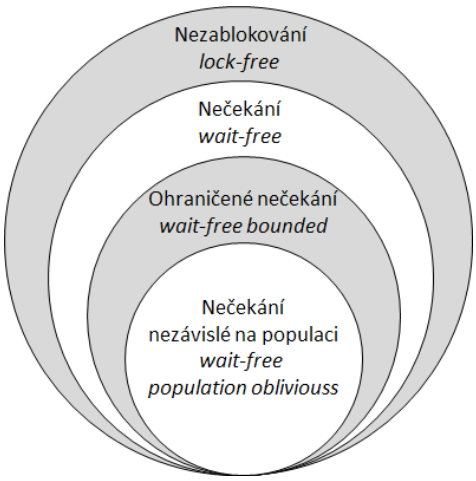
- **Vlastnost ohraničeného nečekání** (wait-free bounded) – nezávislá vlastnost. Každá funkce, která má vlastnost ohraničeného nečekání má i vlastnost nečekání, obráceně tato podmínka nemusí nutně platit.

  Definice: Funkce je ohraničeně nečekající, pokud garantuje, že každé její volání bude dokončeno v konečném, ohraničeném počtu kroků. Toto ohraničení může záviset na počtu volajících vláken.

- **Vlastnost na populaci nezávislého nečekání** (wait-free population oblivious) – nezávislá vlastnost. Volání funkce s touto vlastností je dokončeno v konečném, ohraničeném počtu kroků, a to bez ohledu na množství vláken (populaci), která funkci volají. Každá funkce, která má vlastnost na populaci nezávislého nečekání má vlastnost ohraničeného nečekání.

  Definice: Funkce, která je nečekající a jejíž výkon (nutný počet vykonaných kroků k dokončení) nezávisí na množství aktivně volaných vláken, je na populaci nezávisle nečekající.

Na obr. 6 jsou zobrazeny vztahy množin funkcí, které mají výše představené vlastnosti. V praxi je mezi nečekajícími a ohraničeně nečekajícími funkcemi jen malý rozdíl (Michael, 2015). Datové struktury, které by byly ohraničené, a přitom by zároveň byly na populaci nezávislé, neexistují. Z ohraničenosti totiž plyne počet kroků, které funkce potřebuje ke svému vykonání v nejhorším možném případě, který odpovídá nějaké maximálně možné populaci, to je v rozporu s požadavkem neomezené populace.



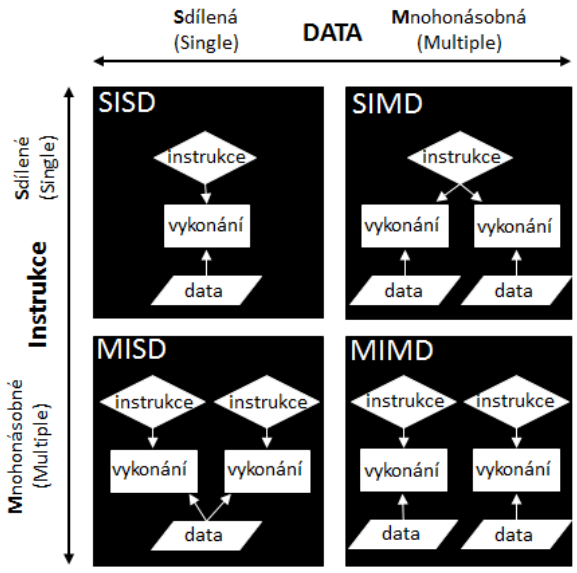
Obrázek 6 – Vztah množin funkcí splňujících různé podmínky pokračování (progress conditions), zdroj: Autor

3.3 Paralelní aplikace

3.3.1 Flynova taxonomie paralelních aplikací

Rozdělení architektur výpočetních systémů podle způsobu hardwarového zpracování instrukcí a práce s daty je možné provést podle Flynnovy taxonomie (Flynn, 1966). Tato taxonomie vychází ze dvou různých charakteristik – způsobu datového toku a způsobu kontroly toku instrukcí. Datový i kontrolní tok mohou být buďto nezávislé nebo jsou naopak sdílené. Z možných kombinací těchto vlastností vychází Flynnova taxonomie. Jde o obecně uznávanou klasifikaci, ze které vycházejí různé možnosti paralelního zpracování výpočtů, jak je popsáno např. v (Almasi & Gottlieb, 1989). Flynnova taxonomie rozděluje architektury výpočetních systémů do následujících kategorií, viz obr. 4:

- **SISD** (Single Instruction Single Data) – toto je klasická architektura procesorů, odpovídá von Neumanově architektuře. V jeden moment je zpracovávaná pouze jedna instrukce, která pracuje nad jednou datovou položkou.
- **SIMD** (Single Instruction Multiple Data) – architektura, ve které je použito více výpočetních jednotek (procesorů nebo jader procesorů), každá výpočetní jednotka pracuje nad svojí vlastní datovou položkou. Výpočetní jednotky ale používají v jednom výpočetním kroku pouze jednu shodnou výpočetní instrukci.
- **MISD** (Multiple Instructions Single Data) – tato architektura fakticky neexistuje, je uváděna pouze pro kompletnost výčtu (nicméně tato architektura by mohla odpovídat záměrnému redundantnímu zpracování stejných dat pro některé bezpečnostně-kritické aplikace).
- **MIMD** (Multiple Instructions Multiple Data) – nejčastěji používaná architektura pro paralelní výpočty, více výpočetních jednotek pracuje nezávisle nad oddělenými daty.



Obrázek 7 - Flynnova taxonomie architektury výpočetních systémů, zdroj: Autor

Z pohledu softwarových aplikací lze paralelní úlohy rozdělit podle toho, jak jednotlivé části paralelní aplikace mezi sebou spolupracují (Foster 1995):

- **SPMD** (Single Program Multiple Data) – tyto úlohy, typicky numerické výpočty, používají jednu kopii programu na všech použitých výpočetních jednotkách, ale zpracovávají během plnění svých dílčích úloh různá data.
- **MPMD** (Multiple Program Multiple Data) – tyto úlohy používají na výpočetních jednotkách různé programy zpracovávající různá data. Tento typ úloh je možné dále rozdělit na úlohy Master-Worker. Ty jsou charakteristické existencí řídícího procesu (master), který řídí ostatní procesy (workers) a zpracovává mezivýsledky připravené ostatními procesy. Druhým typem MPMD úloh je úloha „Coupled Multiple Analyses“. Ta se skládá z paralelních úloh, jenž řeší nezávislé aspekty daného problému.

3.3.2 Výkonnostní metriky paralelních výpočtů

Návrh efektivního paralelního zpracování výpočtů sleduje cíle ve dvou oblastech:

- **výkon** – možnost snížit čas potřebný k dokončení výpočtu zvýšením výpočetních zdrojů,
- **škálovatelnost** – možnost zvýšit výpočetní výkon při zvyšujícím se rozsahu řešené úlohy.

Limity, které tyto cíle omezují, jsou buďto architektonické nebo algoritmické povahy. Architektonické limity vycházení z hardwarových technologií, na kterých jsou založeny fyzické výpočetních zdroje (např. přenosové zpoždění signálů, takt výpočetních jader, šířka komunikačních sběrnic, velikost paměti). Limity algoritmické vycházejí z povahy výpočtů a použitého kódu (možnosti paralelizace – sekvenční versus paralelně zpracovaný kód, velikost komunikační zátěže, možnosti synchronizace, granularita paralelně zpracovaných úkolů).

Aby bylo možné kvantifikovat, jak dobře jsou cíle paralelního zpracování výpočtů naplněny, je třeba použít vhodné metriky ukazující efektivitu paralelních výpočtů. Tyto metriky jsou popsány např. v přehledové studii (Sahni & Thanvantri, 1995). Metriky je možné rozdělit do dvou tříd:

- **Výkonnostní metriky pro výpočetní jednotky** – jsou metriky zabývající se výkonem fyzických výpočetních jednotek (procesorů). Tyto metriky užívají různé formy počtu operací za jednotku času. Běžně používané metriky tohoto typu jsou:

  MIPS – miliony instrukcí za sekundu,

  FLOPS – počet operací v plovoucí řádové čárce za sekundu,

  SPECint/SPECfp – standardizované porovnání výkonů s daty typu int/float (Standard Performance Evaluation Corporation),

Whetstone/Dhrystone – speciální metriky pro měření výkonu procesorů s daty typu float/int.

- **Výkonnostní metriky pro paralelní aplikace** – jsou postaveny na porovnání času běhu aplikací při jejich paralelním běhu na více výpočetních jednotkách a při běhu na jednom procesoru. Mezi užívané metriky této třídy patří:
    - *Zrychlení* – měří poměr mezi dobou běhu aplikace při sekvenčním zpracování a při paralelním zpracování.
    - *Efektivita* – sleduje míru využití výpočetních prostředků, měří poměr mezi výkonností a množstvím výpočetních prostředků (např. počtem výpočetních jednotek).
    - *Redundance* – navýšení množství výpočtů při paralelním zpracování, určuje ji poměr mezi počtem operací provedených při paralelním zpracování a počtem operací při provedení toho samého výpočtu při sekvenčním zpracování.
    - *Utilizace* – charakterizuje míru žádoucího využití výpočetních prostředků, je kvantifikována poměrem výpočetní kapacity, která byla při výpočtu použita, a celkové dostupné výpočetní kapacity.
    - *Kvalita* – metrika pro posouzení relevance užití paralelizace výpočtů v konkrétní aplikaci.

V tabulce 1 jsou jednotlivé výkonnostní metriky pro paralelní aplikace shrnuty spolu s ilustrativními hodnotami pro modelový výpočet trvající při sekvenčním zpracování na jednom procesoru 1 sekundu při provedení 1000 výpočetních operací.

Tabulka 1 - Výkonnostní metriky pro paralelní aplikace, zdroj: Autor

Metrika	Vztah	1 CPU	2 CPU	4 CPU	8 CPU	16 CPU
Doba běhu na p CPUs [ms]	T(p)	1000	520	280	160	100
Zrychlení	S(p) $S(p) = \frac{T(1)}{T(p)}$	1	1,92	3,57	6,25	10
Efektivita	E(p) $E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \cdot T(p)}$	1	0,96	0,89	0,78	0,63
Počet operací při paralelním zpracování	O(p)	10000	10250	11000	12250	15000

Redundance	R(p)	$R(p) = \frac{O(p)}{O(1)}$	1	1,03	1,10	1,23	1,50
Utilizace	U(p)	$U(p) = R(p) \cdot E(p)$	1	0,99	0,98	0,96	0,95
Kvalita	Q(p)	$Q(p) = \frac{S(p) \cdot E(p)}{R(p)}$	1	1,79	2,89	3,96	4,20

Důležitým aspektem při návrhu paralelních aplikací je předpověď, jaké je potenciální, teoretické zrychlení, kterého je možné při paralelním zpracování výpočtu dosáhnout. Vztahy pro tyto předpovědi popisuje několik zákonů, které budou představeny dále:

- Amdahlův zákon
- Gustavson-Barisův zákon
- Karp-Flattův zákon

3.3.3 Amdahlův zákon – pesimistický pohled na zrychlení při paralelizaci

Gene Myron Amdahl v roce 1967 při studiu výkonnosti aplikací pro mainfraimové počítačové systémy formuloval ve své práci (Amdahl 1967) teoretický vztah pro možné zrychlení výpočtů na základě vylepšení výpočetních prostředků:

$$S(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{1}$$

Kde S je teoretické zrychlení běhu celého programu, s je zrychlení té části programu, která využívá vylepšené výpočetní prostředky, p je podíl doby běhu té části programu, která může profitovat z vylepšených výpočetních prostředků a celkové doby běhu programu (před vylepšením). Z tohoto vztahu plyne ihned nepřekročitelné omezení teoretického zrychlení. Omezení je dáno tou částí programu, která nedokáže využít vylepšených výpočetních prostředků:

$$\lim_{s \to \infty} S(s) = \frac{1}{1 - p} \tag{2}$$

Tento vztah lze přeformulovat do podoby popisující možné zrychlení programu, který je paralelizován. Celkovou dobu běhu programu můžeme rozdělit podle typu operací prováděných během výpočtu do tří kategorií:

- $C(\mathrm{seq})$ operace, které musejí být provedeny sekvenčně, nelze je paralelizovat,
- $C(\mathrm{par})$ operace, které mohou být paralelizovány,

- $C(com)$ dodatečné operace pro realizaci paralelizaci (komunikace, synchronizace).

Pro zrychlení paralelizovaného výpočtu potom platí:

$$S(p) = \frac{T(1)}{T(p)} = \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p} + C(com)} \tag{3}$$

Kde p je počet výpočetních jednotek, mezi které je ta část výpočtu, kterou lze paralelizovat, rozdělena.

Protože $C(com) \geq 0$ platí:

$$S(p) \leq \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p}} \tag{4}$$

Označme f poměr doby běhu programu, která musí být provedena sekvenčně a celkové doby běhu programu:

$$f = \frac{C(seq)}{C(seq) + C(par)}; \qquad 0 \leq f \leq 1 \tag{5}$$

Pro maximální možné zrychlení programu jeho paralelizací potom po zjednodušení dostaneme vztah vyjadřující Amdahlův zákon:

$$\mathbf{S(p) \leq \frac{1}{f + \frac{1 - f}{p}}} \tag{6}$$

Amdahlův zákon předpokládá, že úloha řešená výpočtem má fixní rozsah, který se snažíme uskutečnit co nejrychleji. Zákon pak poskytuje horní odhad dosažitelného zrychlení při použití paralelizace. Zrychlení S nemůže být nikdy větší než:

$$S \leq \frac{1}{f} \tag{7}$$

Předpověď celkového zrychlení paralelizovaných výpočtů fixního rozsahu je podle Amdahlova zákona velmi pesimistická. Pokud např. sekvenční část výpočtu tvoří 10 % celkové doby běhu programu, potom maximální teoretické zrychlení bude 10. Prakticky je potom jedno, zda se na výpočtu bude podílet 100 výpočetních jednotek poskytujících zrychlení 9,17 nebo 1000 výpočetních jednotek dosahujících zrychlení 9,91.

Podle Amdahlova zákona se nevyplatí uvažovat o paralelizaci programů, pokud je poměr operací, které musí být provedeny sekvenčně, vůči těm, které lze provádět paralelně, vysoký. Důvod pro takto pesimistickou předpověď leží právě v oné podmínce fixního rozsahu úlohy. Amdahlův zákon totiž ignoruje přínosy paralelizace při zvětšujícím se rozsahu úlohy. Ve skutečnosti ale dochází k žádoucímu jevu, kterému se říká Amdahlův efekt. Při zvětšování rozsahu úlohy roste doba běhu programu

potřebná na samotné výpočty (které se mohou paralelizovat) rychleji než doba běhu operací sekvenční části programu. Proto při daném počtu výpočetních jednotek bývá celkové zrychlení rostoucí funkcí rozsahu úlohy:

$$\text{Rozsah}(\text{úloha 1}) < \text{Rozsah}(\text{úloha 2}) \;\rightarrow\; S_1(\text{úloha 1}) < S_2(\text{úloha 2}) \tag{8}$$

Síla Amdahlova efektu záleží na konkrétním návrhu paralelního algoritmu. Pro rozumně navržené paralelní algoritmy platí, že paralelní režie má asymptoticky nižší složitost než výpočet paralelizované části.

3.3.4 Gustafson-Barsisův zákon – vliv rozsahu úlohy

Vliv rozsahu úlohy na celkové zrychlení při paralelizaci zahrnul do svých úvah John Gustafson v roce 1988. Gustafson pozoroval, že skutečné zrychlení při řešení reálných úloh je větší, než předpovídal Amdahlův zákon. Při formulaci vztahů pro celkové zrychlení paralelizovaného programu stanovil jako konstantní proměnnou celkový čas, potřebný na dokončení úlohy (místo rozsahu úlohy). Gustafson předpokládal, že při použití vyššího počtu výpočetních jednotek může vyřešit ve stejném čase stejnou úlohu přesněji, něž při použití menšího počtu výpočetních jednotek. Gustafson prezentoval tento přístup ve své práci (Gustafson, 1988).

Opět rozdělíme dobu běhu programu na sekvenční a paralelní část. Pro celkové zrychlení při rozdělení té části výpočtu, kterou lze paralelizovat, na p výpočetních jednotek, platí:

$$S(p) \le \frac{C(\text{seq}) + C(\text{par})}{C(\text{seq}) + \frac{C(\text{par})}{p}} \tag{9}$$

Pokud označíme f poměr doby běhu té části výpočtu, která musí být provedena sekvenčně a celkové doby běhu výpočtu, pak je $(1 - f)$ poměr doby paralelního výpočtu vůči celkové době výpočtu a platí:

$$f = \frac{C(\text{seq})}{C(\text{seq}) + \frac{C(\text{par})}{p}} \quad \rightarrow \quad C(\text{seq}) = f \cdot \left(C(\text{seq}) + \frac{C(\text{par})}{p} \right) \tag{10}$$

a

$$(1 - f) = \frac{\frac{C(\text{par})}{p}}{C(\text{seq}) + \frac{C(\text{par})}{p}} \quad \rightarrow \quad C(\text{par}) = p \cdot (1 - f) \cdot \left(C(\text{seq}) + \frac{C(\text{par})}{p} \right) \tag{11}$$

Potom pro celkové zrychlení platí:

$$(p) \le \frac{\left(f + p \cdot (1 - f) \right) \cdot \left(C(\text{seq}) + \frac{C(\text{par})}{p} \right)}{\left(C(\text{seq}) + \frac{C(\text{par})}{p} \right)} \tag{12}$$

A po zjednodušení dostaneme vztah vyjadřující Gustafson-Barsisův zákon:

$$\mathbf{S(p) \leq p + f \cdot (1 - p)} \tag{13}$$

Pokud Amdahlův zákon řeší, jakého zrychlení je možné dosáhnout při paralelizaci daného sekvenčního programu na p výpočetních jednotkách, pak Gustafson-Barsisův zákon přistupuje k výpočtu zrychlení přesně opačně – pro paralelizovaný program řeší, jak dlouho by takový program běžel jako sekvenční, za použití pouze jedné výpočetní jednotky.

3.3.5 Karp-Flattův zákon – vliv komunikační složky

Tato metrika, popsaná v (Karp & Flatt, 1990), je zajímavá zejména proto, že se zabývá vlivem výkonových nákladů na komunikaci, synchronizaci a počáteční inicializaci paralelního výpočtu. Opět předpokládejme, že výpočet se skládá z části sekvenční a z té, kterou lze zpracovat paralelně, a že platí vztahy:

$$\begin{aligned} \mathrm{T(1)} &= \mathrm{C(seq) + C(par)} \\ \mathrm{T(p)} &= \mathrm{C(seq)} + \frac{\mathrm{C(par)}}{\mathrm{p}} + \mathrm{C(com)} \end{aligned} \tag{14}$$

Dále nechť e je experimentálně určená sekvenční část při paralelním zpracování výpočtu:

$$\mathrm{e} = \frac{\mathrm{C(seq)}}{\mathrm{T(1)}} = \frac{\mathrm{C(seq)}}{\mathrm{C(seq) + C(par)}} \tag{15}$$

dostaneme:

$$\begin{aligned} \mathrm{C(seq)} &= \mathrm{e \cdot T(1)} \\ \mathrm{C(par)} &= \mathrm{(1 - e) \cdot T(1)} \end{aligned} \tag{16}$$

Předpokládejme nyní, že část operací potřebných na komunikaci C(com) můžeme zanedbat, potom bude platit:

$$\mathrm{T(p)} = \mathrm{e \cdot T(1)} + \frac{\mathrm{(1 - e) \cdot T(1)}}{\mathrm{p}} \tag{17}$$

Z definice zrychlení plyne vztah pro T (1):

$$\mathrm{S(p)} = \frac{\mathrm{T(1)}}{\mathrm{T(p)}} \quad \rightarrow \quad \mathrm{T(1) = S(p) \cdot T(p)} \tag{18}$$

odtud již dostáváme:

$$\mathrm{T(p)} = \mathrm{e \cdot S(p) \cdot T(p)} + \frac{\mathrm{(1 - e) \cdot S(p) \cdot T(p)}}{\mathrm{p}} \tag{19}$$

A po zjednodušení dostáváme vztah pro Karp-Flattův zákon:

$$\mathbf{e} = \frac{\frac{\mathbf{1}}{\mathbf{S(p)}} - \frac{\mathbf{1}}{\mathbf{p}}}{\mathbf{1} - \frac{\mathbf{1}}{\mathbf{p}}} \tag{20}$$

Čím je hodnota Karp-Flattovy metriky e nižší, tím je paralelizace výpočtu lepší.

Zanedbání vlivu operací potřebných pro komunikaci při paralelním zpracování výpočtu C(com), při odvození vztahu pro metriku e dává možnost zpětně určit vliv komunikace na konečný pokles výkonu paralelizovaného výpočtu.

Pokud uvažujeme efektivitu paralelizace aplikace jako klesající funkci počtu výpočetních jednotek, Karp-Flattova metrika umožní posoudit roli komunikační složky C(com) v tomto poklesu takto:

- Pokud je hodnota metriky e konstantní s rostoucím počtem výpočetních jednotek, potom je zřejmé, že složka C(com) je také konstantní, a proto musí být pokles výkonu způsoben zatím nevyužitou možností paralelizace v dané aplikaci.
- Pokud hodnota metriky e roste s rostoucím počtem výpočetních jednotek, potom musí být pokles výkonu způsoben složkou komunikačních operací C(com), což indikuje příliš velkou zátěž při obsluze paralelizace.

Karp-Flattova metrika tedy umožňuje detekci zdroje neefektivity.

3.3.6 Paralelní zpracování výpočtů – překážky a úkoly

Využití paralelního zpracování pro urychlení výpočtů přináší překážky, které je třeba překonat, aby paralelizmus přinesl zamýšlené benefity. Překážky spočívají v nutnosti splnit dobře známé úkoly, které vycházejí ze zvyšujícího se počtu výpočetních jednotek (processing elements – PE). V přehledové studii programovacích modelů pro vysoko-úrovňové paralelní programování (Belikov at al., 2013) autoři představují 4 kategorie překážek paralelního zpravování, které mají obecný charakter, a které je třeba zvážit při jakémkoli návrhu paralelního zpracování výpočtů:

- **Komplexita paralelního návrhu a implementace** – navržení, vytvoření a odladění programu, který umožňuje paralelní výpočty je mnohem složitější, než je tomu u sekvenční verze programu se stejnou funkcionalitou. Komplexitu paralelní verze programu zvyšují požadavky v těchto oblastech:

- *Koordinace* – koordinační aspekty zahrnující např. volbu způsobu dekompozice výpočtů, jejich mapování na výpočetní jednotky, vzájemnou komunikaci a synchronizaci mezi paralelně zpracovávanými částmi.
- *Přenositelnost* – pro plné využití paralelně pracujících programů je třeba řešit otázku přenositelnosti řešení mezi různými architekturami. A to jak přenositelnost funkční, tak přenositelnost výkonnostní.
- *Produktivita* – implementace paralelního algoritmu zahrnuje práci na úrovni, která je bližší použitému hardwaru. Strukturované, objektové a deklarativní programovací přístupy jsou při návrhu a implementaci paralelních aplikací značně omezeny. Obtížnější je také fáze verifikace a testování. Lze říci, že programování paralelních aplikací vyžaduje značně odlišný způsob myšlení, než je tomu u návrhu a implementace standardních sekvenčních programů.
- *Škálovatelnost* – požadavkem kladeným na paralelní aplikaci je možnost využít zvyšujících se výpočetní prostředků (paměť, počet výpočetních jednotek) pro efektivní řešení zvětšující se úlohy. Některé problémy jsou škálovatelné snadno, u jiných lze škálovatelnosti dosáhnut jen za cenu zvyšující se komplexnosti celé aplikace. V takových případech hrají svou roli většinou zvyšující se výpočetní náklady na komunikaci a synchronizaci.

- **Automatizované řízení paralelního zpracování** – paralelizace řešení úlohy má význam pouze tehdy, pokud výpočetní náklady na paralelní zpracování jsou nižší, než jsou přínosy paralelních výpočtů v porovnání se sekvenčním zpracováním. Rozdělování úlohy na dílčí části, které se mají paralelně zpracovat, a řízení tohoto zpracování je třeba automatizovat. Řešení problému je většinou zapouzdřeno v platformách pro paralelní výpočty pomocí aplikačních kontextů, front úkolů a programových jader (elementárních úkolů pro výpočetní jednotky).
- **Víceúrovňová paralelizace** – současné platformy pro heterogenní systémy a paralelní výpočty používají hierarchickou architekturu (např. MPI, OpenMP, CUDA), která nabízí různé programovací modely pro různé úrovně hierarchie, aby bylo možné využít všechny výhody sdílené či distribuované paměti a masivní datové paralelizace. Programovací modely jsou nízko-úrovňové a nejednotné pro různé druhy výpočetních jednotek. Představují tak velkou zátěž v oblasti produktivity vývojářů a zároveň překážku v přenositelnosti mezi různými hardwarovými architekturami. Alternativou je snaha o unifikaci více úrovňových paralelních programovacích modelů pro více druhů výpočetních jednotek jako jsou procesory či grafické karty (např. platformy OpenCL nebo OmpSs). Programovací moduly zabezpečují funkční přenositelnost, na druhou stranu ale nechávají mnoho koordinačních rozhodnutí na programátorech.

- **Paralelní datové struktury** – mnoho tradičně používaných datových struktur je přirozeně sekvenčních a také vyžadují sekvenční zpracování (např. spojový seznam). Sekvenční části programu podle Amdahlova zákona představují omezení horní hranice možného zrychlení při paralelním zpracování některé části programu, jak je popsáno v kapitole 2.2.1. Použití datových struktur, které umožňují paralelní zpracování, je tedy klíčovou podmínkou pro efektivně pracující paralelní aplikaci. Klíčovým úkolem je navrhnout pro řešení problému implicitně paralelní datové struktury a odstranit koordinační problémy. Mnoho algoritmů lze přepsat do podoby vhodné pro paralelní zpracování použitím stromových datových struktur, které sami o sobě paralelizaci umožňují, jak je demonstrováno v (Steele, 2009).

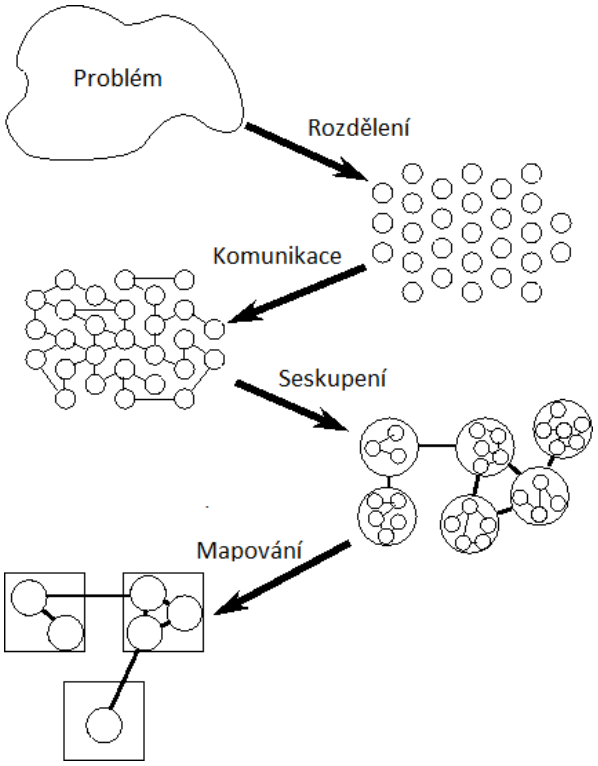
3.4 Fosterova metodika návrhu paralelních aplikací

Návrh správně a efektivně fungujících paralelních programů je složitý proces, který lze jen těžko úspěšně realizovat pouze za použití znalostí klasické algoritmizace a intuice. Jako základ vytvoření metodického postupu pro návrh paralelního zpracování výpočtů v multi-agentových systémech bude použita Fosterova metodika návrhu paralelních programů (Foster, 1995). Jedná se o obecně použitelnou a ve svých krocích velice konkrétní a uchopitelnou metodikou.

Paralelizaci řešení většiny problémů je možné provést několika odlišnými způsoby. Optimální paralelní řešení často neodpovídá původnímu návrhu sekvenčního zpracování. Pomocí metodického přístupu podle Fostera lze proces paralelního návrhu rozdělit na tyto čtyři fáze, viz obr. 8:

1. *Rozdělení* – rozdělení paralelizované úlohy na malé elementární úkoly.
2. *Komunikace* – identifikace komunikace, kterou je třeba vykonávat při paralelním způsobu zpracování elementárních úkolů.
3. *Seskupení* – návrh optimální struktury, kombinující elementární úkoly do skupin s ohledem na nutnou komunikaci, požadavky na výkon, dostupné výpočetní prostředky a náročnost implementace.
4. *Mapování* – návrh způsobu rozdělení úkolů mezi jednotlivé výpočetní jednotky, snahou je nalézt optimální kompromis mezi maximalizací využití výkonu jednotlivých výpočetních jednotek a minimalizací nákladů na komunikaci a celkovou režii paralelizace.

Pro Fosterovu metodiku se používá označení PCAM (Partitioning, Communication, Agglomeration, Mapping).



Obrázek 8 - Fáze Fosterovy metodiky návrhu paralelní aplikace, zdroj: upraveno podle (Foster, 1995)

3.4.1 Rozdělení

Tato fáze je určena hlavně k tomu, aby byl identifikován veškerý potenciál k možnému paralelizování úlohy. Proto je rozdělení na elementární úlohy prováděno nejprve bez ohledu na další limitace jako je komunikace a možnost technického řešení paralelizace. V této fázi dochází k tzv. jemnému rozdělení (fine-grained).

Rozdělení musí proběhnout ve dvou rovinách – v rovině výpočtů, které se mají provádět a v rovině dat, na kterých se mají výpočty provádět. Dekompozici je možné provádět dvěma přístupy:

- *Doménová dekompozice* – tento přístup je běžnější, nejprve se identifikují data související s řešeným problémem, poté jsou data rozdělena na části o přibližně stejné velikosti, nakonec jsou identifikovány výpočetní úkoly, které je třeba nad těmito daty vykonat.
- *Funkční dekompozice* – tento přístup se nejprve zaměřuje na samotné výpočty, které je třeba provést, cílem je nalézt disjunktní množiny úkolů, nesdílející data potřebná k jejich provedení.

Doménová dekompozice zabývající se primárně daty je základem pro návrh paralelního zpracování. Funkční dekompozice je odlišný a doplňující způsob, jak o problému uvažovat. Je to proces, při kterém je pohled zaměřen primárně na prováděné výpočty místo na data, se kterými se výpočty provádějí. Tento přístup může pomoci identifikovat datové struktury problému, které se pro paralelizaci hodí lépe než struktury, jež je možné identifikovat pomocí doménové dekompozice dat.

Foster jako součást své metodiky pro každou fázi navrhuje v (Foster, 1995) několik kontrolní bodů, jejichž splnění pomůže dosáhnout kvalitního paralelního návrhu. Pro fázi rozdělení jsou to tyto body:

- Počet identifikovaných elementárních úkolů je řádově větší, než je počet dostupných výpočetních jednotek.
- Rozdělení úkolů v sobě neobsahuje výpočty prováděné opakovaně a neobsahuje požadavky na opakované uložení a vyzvednutí dílčích výpočtů.
- Identifikované elementární úkoly mají všechny podobnou velikost.
- Množství elementárních úkolů roste úměrně zvyšující se velikosti řešené úlohy. Ideální je, pokud při rostoucí velikosti úlohy roste počet elementárních úkolů, ne jejich velikost.
- Dobré je identifikovat několik variant rozdělení na elementární úkoly – poskytuje to flexibilitu v dalších fázích návrhu.

3.4.2 Komunikace

Výpočty prováděné jednou úlohou zpravidla potřebují data, která jsou zpracována úlohou jinou. Pokud jsou úlohy zpracovávány odděleně, musí dojít k výměně dat mezi nimi. Tato fáze návrhu paralelního zpracování se zabývá tokem těchto dat a komunikací mezi oddělenými částmi výpočtu. Pokud je předchozí fáze (rozdělení) provedena jako doménová dekompozice s důrazem na data, je návrh komunikace obtížný, k rozdělení totiž došlo na základě hledání disjunktních podmnožin dat. Požadavky na komunikaci potom mohou být nejasné. Při funkční dekompozici je naopak přímočará spojitost mezi disjunktními skupinami úkolů a komunikací, která mezi těmito skupinami musí probíhat.

V rámci identifikace komunikačních vzorců je možné se rozhodovat na základě známých komunikačních kategorií a komunikačních vzorců. Je třeba rozlišit potřeby lokální a globální komunikace. Vzorec lokální komunikace odpovídá výměně dat mezi konkrétní elementární úlohou a omezeným počtem ostatních elementárních úloh, které jsou prostorově dané úloze blízké, zatímco při globální komunikaci musí každá elementární úloha komunikovat s velkým počtem ostatních úloh.

Možnosti paralelizace a potřebné komunikace mezi elementárními úlohami je možné odhalit pomocí dalšího komunikačního vzorce „rozděl a panuj“. Snahou je nalezení nějaké rekurentní struktury v dané úloze. Struktura komunikujících částí úlohy potom může odpovídat např. struktuře stromu.

Pokud prvky, které se účastní komunikace při paralelním zpracování, netvoří nějakou dobře uchopitelnou a popsatelnou strukturu nebo pokud se tato struktura během výpočtu mění, je třeba se na komunikaci podívat mnohem komplexnějším způsobem.

Dalším aspektem komunikace je to, zda požadovaná komunikace je synchronní nebo asynchronní. Pokud poskytovatel i příjemce komunikovaných dat (odděleně zpracované úlohy) přesně vědí, kdy se

potřebná komunikace odehrává, jde o komunikaci synchronní. Pokud naopak může být komunikace vyžadována na základě událostí, jejichž výskyt nelze přesně předpovědět, je třeba komunikace asynchronní. Taková komunikace musí být inicializována tou úlohou, která data v dané chvíli od jiné dílčí úlohy potřebuje a tato úloha vlastnící potřebná data je poskytne až ve chvíli, kdy jsou data skutečně k dispozici.

Podobně jako pro fázi rozdělení existují kontrolní body pro fázi komunikace:

- Míra potřebné komunikace, které se účastní jednotlivé dílčími úlohy, by měla být, pokud možno na stejné úrovni pro všechny tyto úlohy.
- Každá dílčí úloha by měla komunikovat pouze s omezeným počtem jiných dílčích úloh.
- Komunikační operace by měly být realizovatelné souběžně. Pokud tomu tak není, je ohrožena škálovatelnost úlohy. Pro odhalení možností souběhu komunikace je vhodná technika hledání rekurzivní struktury v úloze pomocí vzorce „rozděl a panuj“.

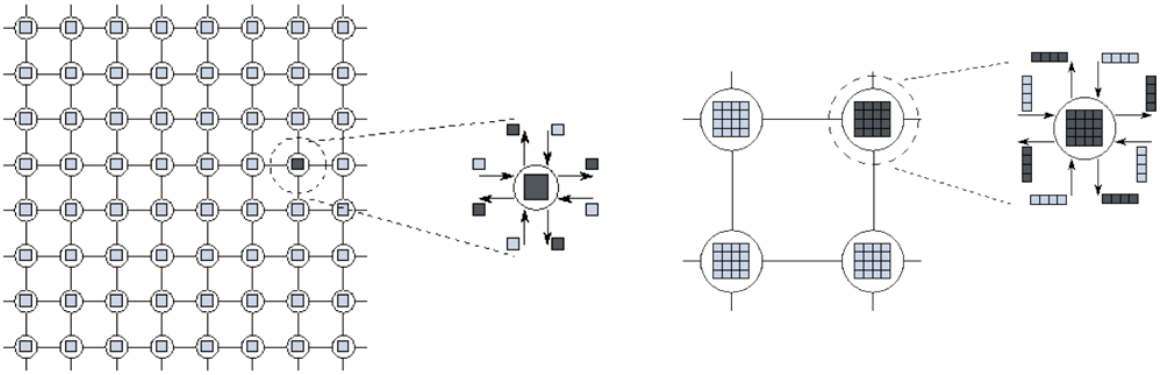
3.4.3 Seskupení

V této fázi se z abstraktních struktur odhalených v prvních dvou fázích musí stát struktura konkrétní, vhodná pro návrh algoritmu, který bude realizován pomocí zamýšlených výpočetních prostředků. Nalezení co největšího počtu dílčích úloh bylo užitečné pro odhalení možností paralelizace, při seskupení je snahou navrhnout správnou granularitu rozdělení problému, která minimalizuje poměr nákladů na komunikaci a množství prováděných výpočtů.

Dalším aspektem jsou náklady na vytváření dílčích úloh. Pokud je dílčích úloh mnoho a náklady na jejich vytváření jsou velké, mohou být veškeré přínosy paralelního zpracování těchto úloh převáženy náklady na vytváření těchto úloh.

Pro efektivní seskupení se využívá tzv. povrchově-objemového efektu (surface-to-scale effect). Jde o to, že při vhodném seskupení dílčích úloh rostou při zvětšujícím se rozsahu celého problému náklady na komunikaci významně pomaleji, než roste objem výpočtů. Tento efekt je stejné povahy jako je Amdahlův efekt představený v kapitole 3.3.3.

Příklad povrchově-objemového efektu je zachycen na obr. 9. Případ (a) zachycuje výpočet rozdělený na 8 x 8 = 64 elementárních úkolů, každá dílčí úloha provádí výpočet pro jednu buňku a komunikuje s osmi sousedními úlohami, celkový počet komunikačních zpráv je v tomto případě tedy 64 x 8 = 256. Případ (b) pracuje se stejnou výchozí úlohou, tentokrát jsou ale elementární úlohy seskupeny, každou skupinu tvoří 4 x 4 = 16 buněk, máme tedy celkem 2 X 2 = 4 dílčí úlohy a počet komunikačních zpráv je v tomto případě pouze 4 x 8 = 32.



(a) (b)

Obrázek 9 - Povrchově-objemový efekt při seskupení, zdroj: upraveno podle (Foster, 1995)

Dalším aspektem návrhové fáze seskupení je přístup k opakovaným výpočtům. Při návrhu je v některých případech výhodnější dát přednost opakujícím se výpočtům (replicating computations) v oddělených úlohách před jediným výpočtem a komunikací výsledku tohoto výpočtu na mnoho jiných výpočetních úloh. Vždy je třeba brát v úvahu poměr nákladů na celkovou komunikaci vůči nákladům na opakovaně provedený výpočet a na škálovatelnost řešení (tj. vliv rozsahu celkové úlohy na počet elementárních úloh, jimž je třeba výsledek centralizovaného výpočtu distribuovat versus náklady na opakovaně prováděný výpočet v elementárních úlohách).

Neméně důležitým pohledem je při návrhu ve fázi seskupení celková cena realizace navrženého řešení. Snaha o co nejefektivnější řešení a co nejvyšší flexibilitu paralelní aplikace vůči zvětšujícímu se rozsahu úlohy nemusí být vždy tím nejlepším přístupem. Může vést k extrémnímu růstu nákladů na vývoj a údržbu takové paralelní aplikace – jednak na samotnou realizaci návrhu a jeho testování, ale také na případné úpravy řešení např. při rozšiřujícím se či měnícím se zadání.

Kontrolní body pro fázi návrhu seskupení jsou následující:

- Seskupení by mělo omezit celkový počet komunikačních úloh správným použitím lokalizace výpočtů (uplatněním povrchově-objemového efektu).
- Pokud při seskupení dochází k požadavkům na replikaci dat pro dílčí paralelizované úlohy, nemělo by to příliš omezit možnosti škálovatelnosti celé aplikace. Např. při zvětšující se velikosti řešeného problému by neměla velikost celkové potřebné paměti růst řádově vyšším tempem.
- Pokud dochází při seskupení k růstu velikosti úkolů řešených paralelně, měly by rozdílné typy úkolů zachovat vzájemně porovnatelnou časovou náročnost, aby nedocházelo při paralelním zpracování rozdílně náročných úkolů ke zbytečným synchronizačním prostojům (čekání jednodušších úkolů na výsledky těch nesouměřitelně složitějších).

- Pro zachování škálovatelnosti by celkový počet dílčích, paralelně zpracovávaných úkolů měl i po seskupení stále růst proporcionálně stejně jako velikost celkového problému.
- Správná strategie seskupení by měla být cestou, jak snížit velké náklady na převod sekvenčních programů na jejich paralelní verze při příliš jemném rozdělení úlohy na její dílčí části.

3.4.4 Mapování

Ve fázi mapování jde o specifikaci toho, jak jsou jednotlivé dílčí úlohy přiřazovány na dostupné výpočetní jednotky ke svému paralelnímu zpracování. Cílem je navrhnout takový algoritmus přiřazování úloh výpočetním jednotkám, který by minimalizoval celkový čas výpočtu. Existují dvě základní strategie, které je možné při mapování uplatnit:

1. Přiřazovat dílčí úlohy, které mohou být provedeny nezávisle (nepotřebují spolu komunikovat) na různé, oddělené výpočetní jednotky.
2. Přiřazovat dílčí úlohy, které spolu musejí komunikovat na jednu, společnou výpočetní jednotku.

Tyto dvě strategie mohou být za některých okolností ve vzájemném konfliktu, potom je nutné nalézt kompromisní řešení. Celkový počet úloh, které lze umístit na jednu výpočetní jednotku je také limitujícím faktorem, který je třeba při mapování vzít v úvahu. Nejkomplexnější řešení vyžadují situace, kdy se počet dílčích úloh i počet výpočetních jednotek dynamicky mění za běhu programu.

Problém nalezení optimálního řešení pro mapování dílčích úloh na dostupné výpočetní jednotky je známý NP-kompletní problém. To znamená, že tento problém je algoritmicky neřešitelný. Existují ale známá řešení používající různé heuristiky pro různé třídy paralelizovaných úloh, které dokáží mapování efektivně řešit.

Rekurzivní bisekce (rekurzivní půlení) se snaží rozdělit problémovou doménu na sub-domény přibližně stejné výpočetní náročnosti při snaze o minimalizaci celkových nákladů na komunikaci, tj. o minimalizaci celkového počtu komunikačních kanálů, které mezi jednotlivými sub-doménami musejí existovat. Realizace této techniky probíhá metodou rozděl-a-panuj. Problémová doména je rozdělena v jednom rozměru na dvě sub-domény. Řezy sub-domén jsou pak prováděny rekurentně až do doby, než je nalezen potřebný počet sub-domén. Mezi konkrétní techniky rekurzivní bisekce patří např. techniky:

- *Rekurzivní bisekce podle souřadnic* – nejjednodušší způsob dělení, používá se na nepravidelné souřadnicové sítě, doména je rekurentně rozdělována na dvě poloviny, vždy ve směru svého delšího rozměru. Tato metoda je jednoduchá a výpočetně nenáročná. Nevýhodou je to, že metoda neoptimalizuje náklady na komunikaci. Mohou vznikat dlouhé, úzké sub-domény,

které, pokud je intenzita vyžadované lokální mezi-doménové komunikace vysoká, vygenerují více požadavků na komunikaci, než by bylo zapotřebí v samotné nerozdělené nad-doméně.

- *Nevyvážená rekurzivní bisekce* – variace předchozí metody, která se snaží minimalizovat nutnou komunikaci tak, že místo prostého rozpůlení vytváří subdomény s lepším poměrem stran, než měla původní doména.
- *Rekurzivní grafová bisekce* – metoda používaná pro problémy uspořádané do komplexní, nestrukturované sítě, např. konečné sítě elementů. Tato metoda se snaží rekurentně dělit síť na dvě části, které byly spojeny co nejmenším počtem hran.

Pravděpodobnostní mapování je alternativní technikou mapování. Dílčí úlohy jsou přiřazovány na výpočetní jednotky náhodně, při velkém množství dílčích úloh by vytížení výpočetních jednotek mělo být stejné. Výhodou tohoto přístupu je nízká výpočetní náročnost mapování, nevýhodou je neoptimalizovaná komunikace. Přístup se hodí v případech, kdy je celková komunikační zátěž mezi úlohami nízká.

Při cyklickém mapování je každé m-té výpočetní jednotce z celkového počtu n přiřazena pro k = 1, 2, … každá úloha označená číslem k *m. Metoda je výpočetně nenáročná, ale vyžaduje silnou prostorovou lokalizaci dílčích úloh. Místo alokace pouze jedné úlohy je možné použít i cyklickou alokaci definovaných bloků dílčích úloh.

Kontrolní body pro fázi návrhu mapování jsou následující:

- Při centralizované správě přiřazování dílčích úkolů na výpočetní jednotky se tato správa nesmí stát limitujícím faktorem celkového výkonu.
- Při dynamickém mapování musejí být zváženy celkové náklady na vývoj v porovnání s některými jednoduššími strategiemi (pravděpodobnostní a cyklické mapování).
- Při použití pravděpodobnostních metod mapování musí být počet celkových dílčích úloh dostatečně velký, aby metoda byla účinná.
- Při použití cyklického mapování, musí být úloha silně prostorově lokalizovaná (do pravidelné sítě).

3.5 Heterogenní výpočetní systémy pro paralelní aplikace

Zvyšující se komplexita výpočetních systémů, množství interakcí mezi jejich částmi a specifické nároky na výkon si vyžádaly značnou optimalizaci výpočetních zařízení. Optimalizace se u široké škály zařízení dosáhlo specializací. Dnešní výpočetní systémy, které se z většího či menšího počtu takovýchto specializovaných zařízení skládají, se tak stávají přirozeně heterogenními.

Heterogenní systémy umožňují programátorům zvolit si pro jednotlivé typy úloh vhodné výpočetní zařízení tak, aby jeho specifické možnosti byly využity co nejlépe ve prospěch celého systému. Volba je zpravidla založena na typu hlavní zátěže dané aplikace. Některé aplikace jsou náročné na vyhledávání a třídění, jiné vyžadují intenzivní zpracovaní a výměnu dat, další jsou náročné na matematické výpočty. Každý typ aplikace zpravidla běží nejefektivněji při použití specifického typu hardwarové architektury, jak je popsáno v (Asanovic at al., 2006).

V rámci heterogenních výpočetních systémů zaujímá významné místo tzv. GPGPU – General Purpose Computation on Graphic Processor Units. Zatímco moderní procesory disponují jednotkami výpočetních jader, grafické karty poskytují nejen desítky až stovky fyzických jader, ale i další možnosti ještě masivnějšího souběžného zpracování dat, a to díky principům stream procesorů (AMD) či CUDA jader (NVIDIA). V dnešní době se počet stream procesorů a CUDA jader pohybuje na úrovni několika tisíců. Grafické karty se tak stávají excelentními zařízeními pro paralelní výpočty.

3.5.1 Grafické karty jako výpočetní hardware

GPU jsou z pohledu paralelního zpracování výpočtů velmi zajímavým hardwarem, mají totiž mnohem vyšší počet výpočetních jader, než dnešní CPU. Výpočetní jádra GPU mají ale několik omezení. Jsou seskupena do SMP (Streaming Multi-Processor Units) jednotek, které používají SIMD výpočetní model. To znamená, že každá SMP jednotka má stejný program, všechna výpočetní jádra v jedné SMP jednotce vykonávají v jednom výpočetním taktu vždy stejnou instrukci, pouze nad jinými daty (viz kapitola 3.3.1). Při dobře navržených paralelních algoritmech a implicitně paralelních datových strukturách poskytuje architektura GPU obrovský potenciál pro urychlení výpočtů.

Výroba grafických čipů je motivována poptávkou zejména v těchto odvětvích:

- Herní průmysl – bezesporu nejvíce grafických karet se vyrobí a prodá v tomto sektoru, navíc rozvoj herní oblasti umožnil zdokonalení grafických karet pro profesionální a vědecké aplikace. V současné době oba největší výrobci grafických čipů učinili další velký technologický skok v efektivitě a výkonu, a to AMD s architekturou grafických čipů Polaris a Vega (technologie 14 nm) a NVIDIA s architekturou grafických čipů Pascal (technologie 16 nm).
- Profesionální grafické aplikace – speciální aplikace vyžadující vysoký grafický výkon, např. CAD systémy, GIS aplikace, aplikace pro renderování nebo aplikace pro střih a editaci videa.
- Speciální aplikace pro rychlé výpočty – grafické karty speciálně navržené pro rychlé, výpočetně náročné GPGPU aplikace. Na těchto kartách jsou postaveny současné cloudové služby nabízející rychlý výpočetní výkon jako službu, např. Google Accelerated Cloud Computing, používající high-endové GPU, jako jsou AMD FirePro S9300 x2 nebo NVIDIA® Tesla® P100.

Grafické karty ze sektoru herních GPU, zejména z high-endové třídy těchto karet, se dají velice dobře použít pro vědeckou práci, což bude demonstrováno na případu paralelizace výpočtů v masivně multi-agentových simulacích. Zajímavým hlediskem je jistě poměr ceny a výkonu několik let starých modelů high-endových grafických karet. Nové modely vedle výkonu přinášejí i nižší spotřebu. Dalším aspektem je masivní rozšíření výpočetně silných grafických karet, vědeckému využití se nabízí také možnost propojování více grafických karet do větších výpočetních celků. Ve standardních PC sestavách se sběrnicí PCI Express lze pomocí technologií NVIDIA SLI nebo AMD Crossfire propojovat více grafických karet do jediného výpočetního celku.

3.5.2 Platformy pro heterogenní výpočetní systémy

V dnešní době existuje velké množství platforem pro vytváření heterogenní výpočetní systémů. V přehledové studii o heterogenních programovacích technikách (Mittal a Vetter 2015) autoři detailně rozebírají výhody použití rozdílných druhů výpočetních jednotek v rámci jednoho heterogenního výpočetního systému (CPU-GPU). Jsou zde zrevidovány různé přístupy k heterogenním systémům, a to z několika úhlů pohledu: běh programu, použité algoritmy, způsob kompilace, fungování aplikační vrstvy. Studie poskytuje referenční informace o mnoha existujících výzkumných pracích z problémových domén, které využívají následující platformy pro heterogenní výpočty: OpenCL, CUDA, Brook+, ACML, OpenMP, Pthread, Intel MKL, Intel TBB.

Mezi množstvím platforem pro heterogenní výpočetní systémy zaujímají výjimečné místo a jsou nejčastěji používány a vzájemně porovnávány platformy OpenCL a CUDA. Studie ukazují, že výkonnostně jsou obě platformy srovnatelné, a to po celou dobu jejich existence (Fang at al., 2011). Studie (Souza at al., 2013) porovnávající výkonnost obou platforem na paralelní implementaci kryptografického algoritmu Keccak (v roce 2012 vítězný algoritmus soutěže o nový hashovací algoritmus) dochází k závěru, že OpenCL vykazuje s každou svou novou generací podstatná zlepšení.

Platforma OpenCL má proti platformě CUDA několik výhod: širší podporu od výrobců hardwaru, funkční přenositelnosti aplikací mezi různými architekturami a možnost použití široké škály CPU a GPU (CUDA pracuje pouze s NVIDIA grafickými kartami).

3.5.3 OpenCL – otevřená platforma pro heterogenní systémy

OpenCL (Open Computing Language) je otevřený rámec pro vývoj heterogenních systémů, který je vyvíjen neziskovým technologickým konsorciem Khronos Group. OpenCL umožňuje vytváření aplikací běžících na různých hardwarových zařízeních od různých výrobců. Hlavním cílem použití OpenCL je úkolová nebo datová paralelizace na více zařízeních nebo na jednom zařízení s více výpočetními

jednotkami (jádry). OpenCL dovoluje pohlížet na více-jádrové CPU a GPU jako na společně spravovanou výpočetní platformu (Khronos, 2016; Laville at al., 2016).

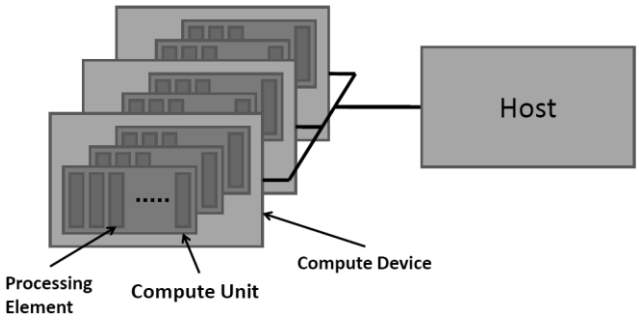
První verze OpenCL byla vydána v roce 2008 a brzy byla přijata hlavními výrobci hardwaru do jejich SDK. Poslední verzí je OpenCL 2.2 z roku 2016. Dnes má OpenCL velkou podporu mnoha výrobců hardware.

OpenCL zapouzdřuje nízko-úrovňové API, které zpřístupňuje hardwarové prostředky grafických karet a poskytuje programátorům nástroje k vytváření GPGPU aplikací pomocí vyšších programovacích jazyků. Pro host-aplikace je v OpenCL používán jazyk C++ (konfigurace OpenCL platformy a zařízení, zprostředkování kompilace programu a kernelů, management paměti a transfer dat z aplikace na vlastní výpočetní zařízení a zpět), pro programování kernelů (programů běžících paralelně na výpočetních jednotkách) je používán jazyk C.

Aplikace vytvořené v OpenCL jsou funkčně přenositelné, je možné zkompilovat je a spustit na různých platformách. Efektivita takových aplikací již nutně přenositelná není. Je to způsobeno tím, že nízko-úrovňová část OpenCL je silně vázána na konkrétní typ hardwaru použité grafické karty.

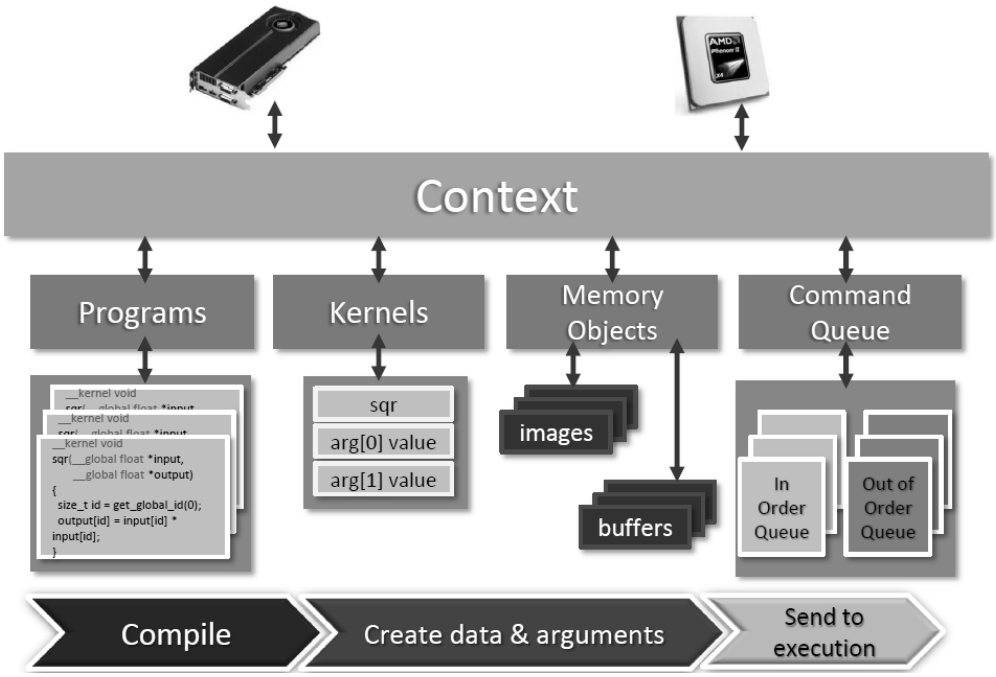
OpenCL specifikace obsahuje čtyři oddělené části – modely, jímž odpovídají příslušné části aplikací, které OpenCL používají (Gaster at al., 2012):

1. *Model platformy a zařízení* – jedna část aplikace, která koordinuje běh celé OpenCL aplikace, se nazývá host aplikace a běží na jednom procesoru platformy. Dalšími částmi jsou OpenCL zařízení (devices). Tyto části odpovídají jednomu či více procesorům nebo grafickým kartám či jiným výpočetním zařízením (obr. 10).



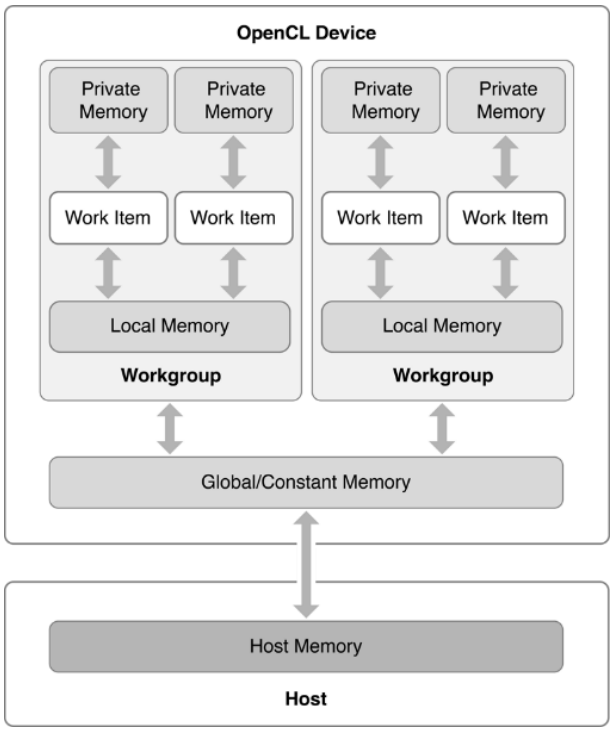
Obrázek 10 - OpenCL host aplikace, platforma a výpočetní zařízení, zdroj: (Khronos, 2016)

2. *Model běhu aplikace* – definuje, jak je v host aplikaci nakonfigurováno OpenCL prostředí pro vlastní běh paralelní aplikace a jak jsou kernely vykonávány na výpočetních zařízeních. OpenCL aplikace definuje kontext, který zprostředkuje interakce host – zařízení (obr. 11).



Obrázek 11 – OpenCL kontext a běh programu, zdroj: (Khronos, 2016)

3. *Model správy paměti* – definuje abstraktní hierarchii paměti používanou aplikací a výpočetními zařízeními. Hierarchie odpovídá tomu, jak je paměť organizována v moderních grafických kartách. Jde o čtyři úrovně paměti: globální, konstantní, lokální a privátní (obr. 12). OpenCL aplikace vyžadují explicitní management paměti. Při přesunu dat mezi host aplikací a zařízeními je nutné nejprve přemístit data z paměti host aplikace do globální paměti zařízení, odtud je možné přenést data do lokální paměti zařízení a pak teprve do privátní paměti zařízení. Podobně zpětnou cestu dat je nutné dodržet.
4. *Model programování* – definuje, jak je prováděna paralelizace naprogramovaného chování na jednotlivých fyzických hardwarových zařízeních. Rozdělení výpočetních úkolů (definovaných kernely) mezi výpočetní jednotky použitých fyzických zařízení je prováděno pro programátora transparentním způsobem. Nízko-úrovňová část programu blízká hardwaru je pomocí OpenCL zapouzdřena.



Obrázek 12 –Model správy paměti OpenCL, zdroj: (Khronos, 2016)

3.6 Multi-agentové modelování a simulace

Agentové modely (Agent-Based Models – ABM, též Individual-Based či Bottom-Up modely) představují rozšířený přístup k modelování komplexních systémů složených z různorodých adaptivních individuálních agentů, kteří spolu interagují v definovaném prostředí. Principy tohoto přístupu jsou popsány v (Macal & North, 2005, 2006, 2010; Railsback & Grimm, 2012; Railsback, 2006; Ulhrmacher & Weyns, 2009). Agentové modelování se úspěšně používá v mnoha problémových doménách, např. v oblasti sociálních simulací (Agent-Based Social Simulations, ABSS), stručné vysvětlení použití v této oblasti z hlediska výpočetních systémů uvádí např. (Gilbert & Troitzsch, 2005; Davidsson, 2002). Existuje mnoho metod a přístupů k vývoji ABS a ABSS, některé jsou popsány v (Grimm at al., 2010; Grimm & Railsback, 2005). V dnešní době existuje nepřeberné množství softwarových systémů, které agentové modelování umožňují (Kravari & Bassiliades, 2015).

Při návrhu multi-agentových modelů se využívají principy objektového návrhu programů (OOP) a také principy agentově orientovaného návrhu programů (AOP), které z principů objektového návrhu vycházejí. Jde především o tyto dva principy:

- *Autonomie* – agenti jsou autonomními entitami, které zapouzdřují svůj stav, chování a způsob, jakým je toto chování ovládáno.

- *Interakce* – vzájemné interakce mezi agenty jsou hlavním faktorem při návrhu agentového systému. Agentový systém je navržen jako dynamická množina agentů, kteří spolu interagují a spolupracují buďto prostřednictvím výměny zpráv nebo zprostředkovaně na základě aktivit a vnímání prostředí, ve kterém jsou umístěni.

Agentově orientovaný návrh je postavený na základech vybudovaných prací (Shoham, 1993). V této práci byly představeny principy agentově-orientovaného programování jako evoluce objektově orientovaného paradigmatu směřující k lepšímu uchopení kognitivních a sociálních schopností agentů. Agenti jsou vnímáni jako specializované objekty vybavené mentálními komponentami (domněnky, záměry, cíle) a schopností interakce s ostatními agenty pomocí vysokoúrovňové komunikace. Tato práce se stala základem mnoha současných agentových programovacích jazyků.

Současný stav výzkumu v oblasti návrhu multi-agentových modelů se úzce dotýká také témat ambientní inteligence, rozšířené reality, modelování chytrých prostředí a internetu věcí. Důležitým konceptem, který propojuje výzkumy agentových modelů a rozšířené reality je koncept tzv. zrcadlových světů (mirror word – MW). Jde o koncept, ve kterém se návrh agentových modelů nové generace nesoustředí pouze na model jako samostatně fungující softwarový systém, ale zabývá se i oboustranným propojením fyzického a digitálního světa (Ricci at al., 2015). Tento koncept přináší nový pohled i pro návrh paralelně pracujících multi-agentové systémů. Samotné entity takových systémů dnes paralelní zpracování umožňují (např. v systémech rozšířené reality hrají častou roli agentů chytré telefony uživatelů, tyto telefony dnes představují převážně více-jádrové výpočetní systémy).

3.6.1 Protokol ODD+D

Protokol ODD (Overview – Design concepts – Details) byl vytvořen komunitou zabývající se tzv. individual-based modely. Jedná se o modely, ve kterých entity, které nejsou lidmi, interagují s okolním, většinou nějakým ekologickým systémem. V tomto pojetí se agenti v agent-based modelech od „individuí“ v individual-based modelech odlišují tím, že jde o lidské agenty, přičemž model reprezentuje chování a rozhodování těchto lidských aktérů. Z pohledu možností využití některých myšlenek protokolu ODD při návrhu paralelních multi-agentových systémů je rozlišení individual/agent-based podružné. Záměrem vytvoření protokolu ODD bylo učinit popis modelu snadněji pochopitelným a usnadnit sdílení vytvořených modelů.

Ačkoli původní verze protokolu ODD je zaměřena primárně na modely dynamiky v ekologických systémech, již první revize protokolu se snaží poskytnout standard pro popis všech typů agentových modelů (Grimm et al., 2010). Protokol ODD byl později rozšířen na ODD+D (ODD + Decision) tak, aby umožňoval lépe popsat chování agenta-člověka a jeho rozhodování (Müller at al., 2013).

Protokoly ODD a ODD+D popisují entitu agenta odlišně, než jak to činí objektový návrh. V ODD je popis stavových veličin (atributů) jasně oddělen od popisu chování (metody). Objektový návrh naproti tomu zapouzdřuje obě složky popisu společně do jedné třídy objektů. Tvůrci ODD považovali popis modelu oddělující popis stavu modelovaného systému od dynamiky tohoto systému za čitelnější a snadněji pochopitelný. Kombinovat strukturu ODD+D protokolu s objektovým návrhem modelu je snadné za pomoci UML diagramů.

Výběr ODD+D protokolu jako jedné z komponent uplatněných při vytvoření metodického postupu pro návrh paralelních agentových systémů byl motivován právě touto vlastností ODD+D, odpovídá totiž přesně myšlence doménové a funkční dekompozice, které se uplatňují v kroku rozdělení Fosterovy metodiky návrhu paralelních aplikací. Právě oddělený pohled na popis stavu (data) a na popis chování a rozhodování (funkce) umožňuje snadnější a ucelenější průzkum možností, jak aplikaci paralelizovat. **Systematizovaný popis modelu, který chceme paralelizovat, tak může být za pomoci protokolu ODD+D snadnější a úplnější, pokud jde o identifikaci potenciálu k paralelnímu zpracování daného modelu a jeho výpočtů.**

Struktura protokolu ODD+D je zachycena v tabulce 2. Detailní popis protokolu ODD+D je k dispozici v příloze A, kde jsou také detailně rozepsány specifické elementy protokolu spolu s naváděcími otázkami.

Tabulka 2 – Struktura protokolu ODD+D, zdroj: zpracováno podle (Müller at al. 2013)

Element protokolu	
Popis **Overview**	• Účel • Entity • Stavové veličiny • Rozměry • Procesy a časování
Koncepty návrhu **Design Concepts**	• Teoretické a empirické předpoklady • Individuální rozhodování • Učení • Individuální vnímání a predikce • Interakce • Skupinové chování • Různorodost a nahodilost • Pozorování
Detaily	• Detaily implementace • Inicializace • Vstupní data • Sub-modely

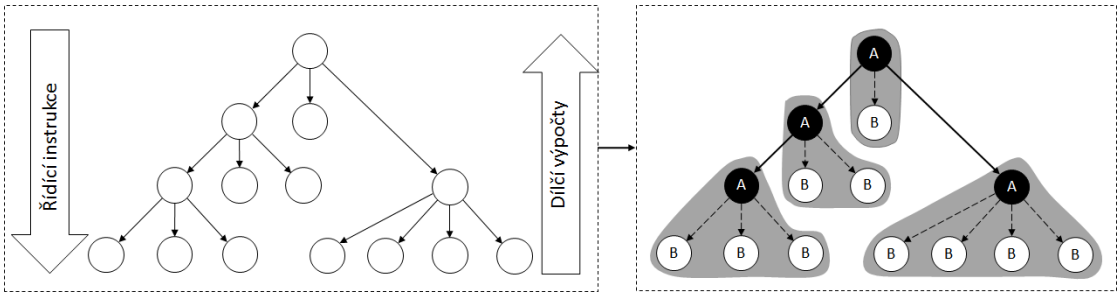
3.6.2 Organizační paradigmata v multi-agentových modelech

Způsob organizace práce jednotlivých agentů v multi-agentových modelech má významný dopad na výkonnostní charakteristiky simulace prováděné nad tímto modelem a na možnosti zvyšovat rychlost běhu simulace pomocí paralelního zpracování. Během let vývoje multi-agentových modelů se objevila celá řada organizačních paradigmat, která mají různé silné a slabé stránky a způsoby využití.

Organizací agentů se rozumí uskupení rolí, vztahů a struktur autorit řídících chování jednotlivých členů. Zvolené organizační paradigma vymezuje možnosti dekompozice řešené úlohy a také z něj vyplývají požadavky na komunikaci jednotlivých agentů (viz. Fosterova metodika návrhu paralelních aplikací, kroky rozdělení a komunikace, viz kapitoly 3.4.1 a 3.4.2). Způsob organizace agentů zásadním způsobem ovlivňuje možnosti a způsob paralelizace výpočtů v multi-agentovém modelu. Organizace a komunikace jsou proto podstatnými složkami paralelního návrhu.

V této kapitole budou představena následující organizační paradigmata: hierarchie, holarchie, koalice, týmy, kongregace, society, federace, trhy, maticové organizace a sdružení (Horling & Lesser, 2004). Nastíněny budou způsoby jejich utváření a údržby, možnosti použití a požadavky na komunikaci. Těmi vlastnostmi je třeba se zabývat z důvodu jejich přímého vlivu na možnosti paralelního návrhu pro model, který dané organizační paradigma používá.

- **Hierarchie** – agenti jsou uspořádáni do stromové struktury, ve které jsou dílčí výstupy produkovány nižšími úrovněmi agentů a distribuovány směrem nahoru k nadřízeným agentům, kteří je agregují a rozhodují o dalším postupu. Řídící instrukce postupují opačným směrem, od nadřízených agentů k podřízeným. Hierarchické uspořádání agentů dovoluje omezit počet interakcí mezi agenty, který je celkově relativně nízký v porovnání s celkovou populací agentů. Toto může být pro paralelní zpracování velkou výhodou.

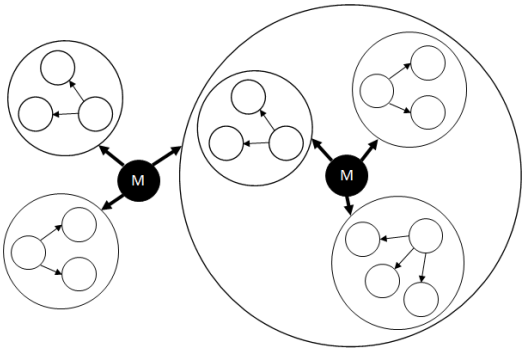


Obrázek 13 - Organizační paradigma Hierarchie, (a) tok řídících instrukcí a dílčích výstupů, (b) distribuce rozhodovací autority a lokalizace problému jako předpoklad pro paralelní zpracování (A – uzel s autoritou, B – uzel produkující dílčí výstupy), zdroj: Autor

Možnosti pro paralelní zpracování výpočtů v hierarchii spočívají v souběžném zpracování dílčích úkolů prováděných v jedné úrovni hierarchie, dílčí výpočty jsou potom předány

nadřízenému uzlu hierarchie. Detailnějšího rozdělení problému na dílčí paralelní úlohy je možné docílit zvýšením úrovně lokalizace problému. Toho lze dosáhnout distribucí rozhodovací autority na různá místa hierarchie (obr. 13).

- **Holarchie** – je seskupením tzv. holonů. Pojem holon (z řeckého holos – celek a on – část) označuje fyzikální systém založený na vnořených, vzájemně podobných strukturách. Holarchiemi jsou např. biologické nebo sociální systémy.

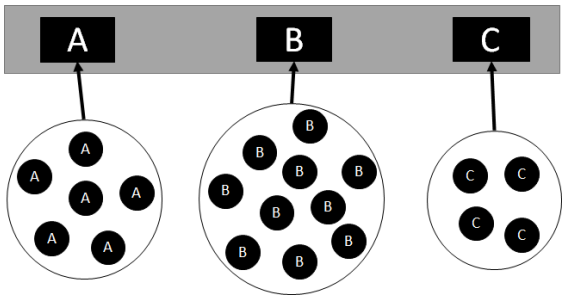


Obrázek 14 - Organizační paradigma Holarchie, jednotlivé holony, propojené pomocí specializovaných holonů mediátorů M., zdroj: Autor

Holarchie jsou charakteristické vysokou mírou autonomie jednotlivých holonů. Agenti uvnitř holonů již zcela autonomní nejsou. V holarchii zpravidla existují holony se specializací mediátorů, které spojují jednotlivé holony dohromady a agregují výsledky, které poskytují (obr. 14). Z hlediska možností paralelního zpracování je možné soustředit se na souběžné zpracování výpočtů jednotlivými holony, komunikační požadavky vycházejí z požadavků na holony-mediátory.

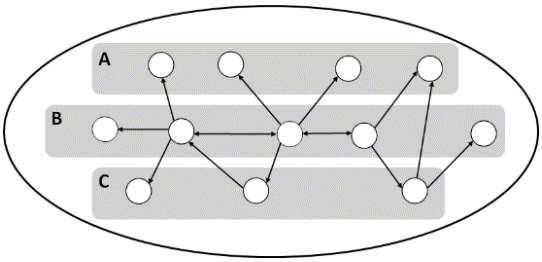
- **Koalice** – jsou nezávislé, krátkodobě existující skupiny agentů, vytvořené z celkové populace agentů k jednomu konkrétnímu účelu. Každý agent přitom sleduje vlastní prospěch. Po splnění svého cíle se koalice rozpadá, agenti se mohou zapojit do jiných koalic a plnit nové úlohy. Na obr. 15 jsou tři různé koalice seskupené k řešení tří rozdílných dílčích úkolů. Organizační paradigma koalice pochází z teorie her a jsou užitečnou strategií pozorovanou v reálném světě a používanou v multi-agentových modelech.

  Při paralelním řešení problému pomocí agentů uspořádaných do koalic je třeba brát v úvahu náklady na vytvoření a řízení jednotlivých koalic.



Obrázek 15 - Organizační paradigma Koalice, dočasná uskupení agentů se společným cílem vyřešit dílčí úlohu A, B nebo C, zdroj: Autor

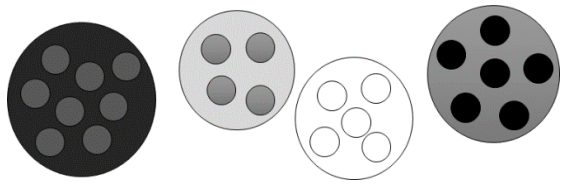
- **Týmy** – podobně, jako v koalicích, i v týmech jsou agenti seskupeni za účelem dosažení konkrétního, společného cíle (obr. 16). Na rozdíl od koalice se členové týmu snaží maximalizovat úspěšnost týmu jako celku. Proto musejí být členové týmu organizováni tak, aby jejich individuální akce byly konzistentní se společným.



Obrázek 16 - Organizační paradigma Tým, struktura týmu a příslušnost členů týmu do skupin pode rolí A, B nebo C, zdroj: Autor

Paralelní zpracování týmových úkolů vyžaduje dekompozici problému na dílčí úkoly podle souboru schopností, nutných k jejich plnění. Agenti, kteří disponují požadovanými schopnostmi, se zaváží splnit dílčí úkol a předat výstup ostatním zainteresovaným agentům.

- **Kongregace** – organizace podobná koalicím a týmům, na rozdíl od týmů a koalic ale vznikají jako dlouho trvající uskupení poskytující při řešení celkového problému nějakou konkrétní schopnost, kterou nelze dosáhnou individuálním nasazením jednotlivých agentů. Jednotlivé kongregace skupin agentů tvoří heterogenní systém koalic, viz obr. 17. Jednotliví agenti nejsou vázáni k fixním úkolům, ale disponují neměnící se sadou schopností, která určuje jejich členství v některé z kongregací.

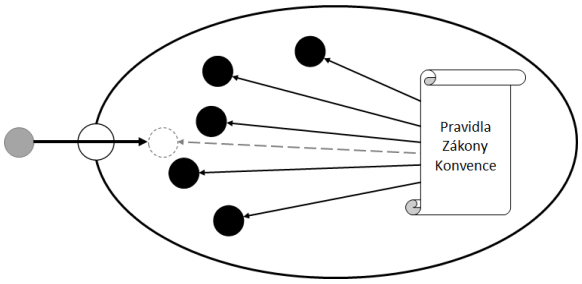


Obrázek 17 - Organizační paradigma kongregace, různé kongregace agentů se společným profilem schopností, zdroj: Autor

Možnosti paralelizace v multi-agentovém modelu, který je organizován pomocí paradigmatu kongregací, leží ve specializaci jednotlivých kongregací a v souběžném plnění jejich dílčích úkolů. Některá z kongregací potom musí zastávat roli koordinátora a agregátora dílčích řešení. Jednotlivé kongregace budou v takovém případě nejspíše uspořádány a organizovány jako hierarchie nebo holarchie.

- **Společnosti –** jsou dlouhodobá uskupení agentů, kteří se zavázali k dodržování dohodnutých pravidel, zákonů či konvencí. Pravidla mohou obsahovat např. akceptovatelné komunikační protokoly nebo soubor omezení chování v rámci dané společnosti, viz obr. 18. Společnost je otevřený systém, do kterého může vstoupit z okolního prostředí jakýkoli agent, který se tato pravidla zaváže také dodržovat. V rámci společnosti mohou být agenti dále organizováni pomocí jiný organizačních paradigmat.

  Při řešení problému pomocí společností může v multi-agentovém modelu existovat několik společností řešících paralelně několik dílčích úloh, odpovídajících veřejné službě, kterou daná společnost poskytuje. Každá společnost může využívat jiný výpočetní prostředek či výpočetní jednotku.

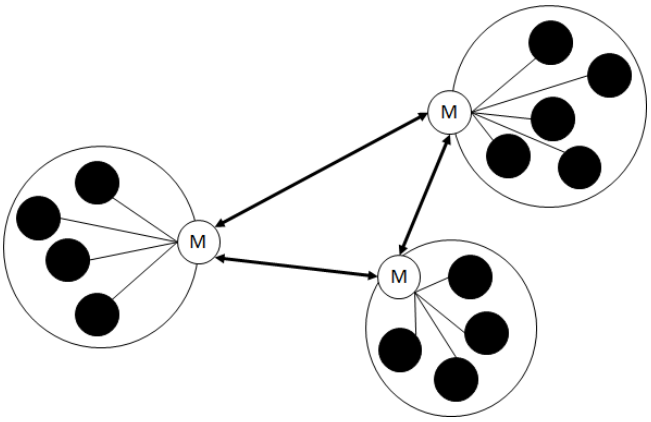


Obrázek 18 - Organizační paradigma společnost, agenti uvnitř společnosti a nový agent vstupující do společnosti, zdroj: Autor

- **Federace** – podobně jako holarchie, pracují s autonomními skupinami agentů. Ve federaci je na rozdíl od holarchie jasně vymezené, jakým způsobem budou takovéto nezávislé skupiny interagovat se svým okolím. K tomuto účelu je určen vždy jeden agent se speciálními schopnostmi zajistit komunikaci s okolními členy federace – tento agent se nazývá mediátor či facilitátor. Agenti jednotlivých skupin komunikují pouze s příslušným mediátorem této skupiny, jak ukazuje obr. 19.

  Mediátoři zprostředkují vyjednávání o přiřazování a plnění dílčích úloh a agregují výsledky poskytované členy své skupiny s výsledky od jiných mediátorů. Tím se sice může zjednodušit celková komunikace (s agenty, kteří jsou skutečnými vykonavateli úloh komunikuje jen jich mediátor), na druhou stranu mediátoři se mohou stát úzkými hrdly federace, která mohou znamenat výkonnostní překážku.

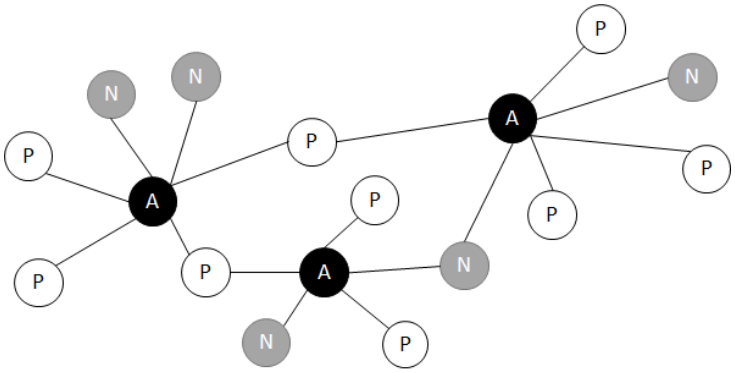
Jednotlivé skupiny federace mohou vystupovat jako nezávislé výpočetní skupiny zpracovávající své úkoly paralelně. Agenti mediátoři pak technicky zastřešují frontu úkolů pro svoji skupinu agentů a poskytují synchronizační služby pro nezávisle prováděné výpočty.



Obrázek 19 - Organizační paradigma federace, skupiny agentů se svými mediátory M, zdroj: Autor

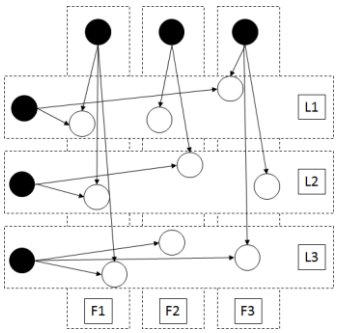
- **Trhy** – v organizaci pomocí paradigmatu trhu vystupují dva druhy agentů – nakupující a prodávající. V některých případech existuje třetí typ agentů, kteří zprostředkují komunikaci mezi nakupujícími a prodávajícími agenty – tzv. aukční agenti, viz obr. 20. Takovéto uspořádání odpovídá vztahu dodavatel-konzument. Trhy s aukčními agenty jsou podobné federacím, v nich rovněž existuje zprostředkovatel komunikace mezi koncovými agenty. Na rozdíl od federace, kde členové jedné skupiny spolupracovali, jsou v organizaci pomocí trhu koncoví agenti v postavení konkurentů.

  Možnost využití více nezávislých výpočetních jednotek a paralelního zpracování nabízí vhodné rozdělení tržních rolí – výpočetní jednotky lze vnímat jako prodávající agenty (poskytují výpočetní výkon) a dílčí úkoly, které je třeba splnit jako vzájemně konkurující si, nakupující agenty (potřebují získat výpočetní čas pro své splnění).



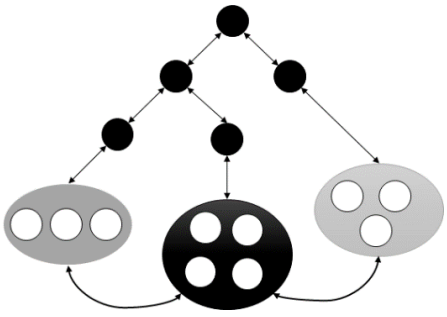
Obrázek 20 - Organizační paradigma trh, nakupující a prodávající agenti N a P, aukční agenti A zprostředkující obchody, zdroj: Autor

- **Maticové organizace** – umožňují, aby jednotliví agenti spadali do kompetence více řídících agentů. Agenti vykonávající dílčí úlohy jsou vnímáni jako omezené zdroje poskytující specifické funkčnosti a schopnosti a jejichž sdílení je nutné koordinovat. Pro efektivní práci při vícenásobném řízení individuálních agentů je třeba přítomnost mechanismu řešícího konflikty v úkolování jednoho agenta a rozhodovací autority, která dokáže takové konflikty vyřešit. Řídící autorita může být uplatňována ve dvou rovinách – liniové rovině (náležení agenta do skupiny např. podle lokality) a funkční rovině (odpovídá specifickým schopnostem agentů plnit určité druhy úkolů), jak je ilustrováno na obr. 21. Maticová organizace je dynamické uspořádání. Vliv konkrétních řídících agentů na výkonné agenty se může v čase měnit podle typu plněných dílčích úloh a profilu dostupných výkonných agentů.



Obrázek 21 - Organizační paradigma maticová organizace, liniové L1-L3 a funkční F1-F3 řídící úrovně, zdroj: Autor

- **Sdružení** – některé typy organizací nelze vymezit jen jedním konkrétním organizačním paradigmatem, ale sestávají ze sdružení různých typů organizace. Jeden druh organizace může být použit pro řízení agentů, jiný pro tok dat. Takovým uspořádáním může být například kombinace hierarchické struktury a skupiny koalic nebo kongregací, viz příklad na obr. 22. Charakteristika sdružení je dána charakteristikami jednotlivých použitých organizačních stylů,



Obrázek 22 - Organizační paradigma sdružení, kombinace hierarchie a s kongregacemi, zdroj: autor

V tabulce 3 jsou shrnuty klíčové vlastnosti, hlavní výhody a nevýhody jednotlivých organizačních paradigmat popsaných v této kapitole:

Tabulka 3 – Shrnutí klíčových vlastností, výhod a nevýhod organizačních paradigmat v multi-agentových systémech, zdroj: upraveno podle (Horling & Lesser, 2004)

Organizační paradigma	Klíčové vlastnosti	Výhody	Nevýhody
Hierarchie	dekompozice	vhodné pro mnoho problémových domén, dobře škálovatelné	hrozba úzkých hrdel a zpoždění
Holarchie	dekompozice s autonomií	autonomie funkčních jednotek	špatně předpověditelná výkonost
Koalice	dynamické uskupení řízené cíli	využití velkého počtu agentů	náklady na vytvoření mohou převážit krátkodobě trvající benefity
Tým	skupinová soudržnost	úkolově centrické, hodí se pro problémy s velkou granularitou	vysoké komunikační náklady
Kongregace	dlouhá životnost, řízené užitečností	umožňuje odhalení agentů podle profilů schopností	silné omezující podmínky na náležení do jednotlivých skupin
Společnost	otevřený systém	veřejně poskytované služby, dobře definovaná společná pravidla	potenciálně příliš komplexní, speciální požadavky na sociální schopnosti agentů
Federace	pojící agenti – mediátoři	doporučující/ zprostředkovatelské/ překladatelské služby	zprostředkující agenti jsou úzkými hrdly celé organizace
Trh	konkurence cenou	efektivní alokace zdrojů	možnost nežádoucího jednání agentů, potenciálně vysoká komplexita alokační úlohy
Maticová organizace	více řídících agentů	sdílení zdrojů	možnost konfliktů v požadavcích na agenty
Sdružení	konkurující si organizace	využití více řídících stylů najednou	možné kolize jednotlivých organizačních paradigmat

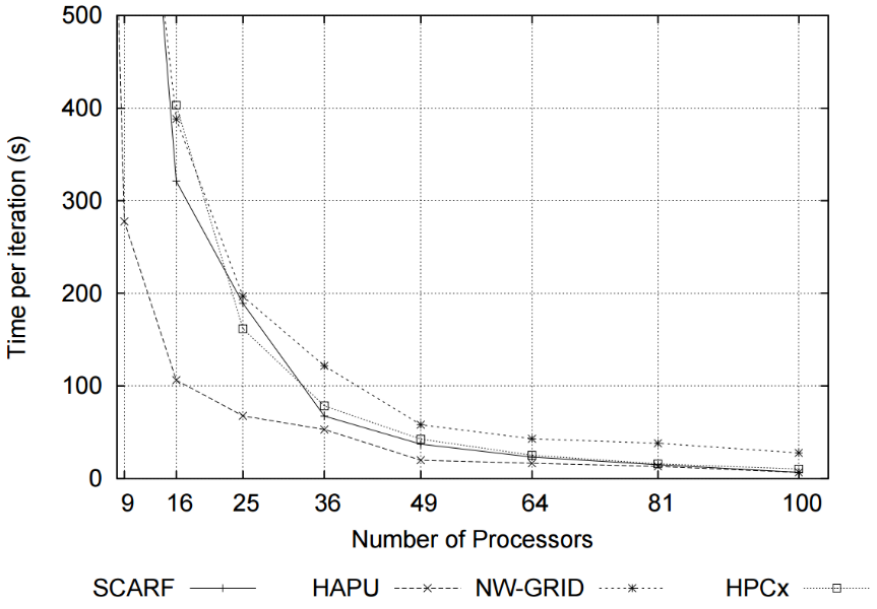
3.6.3 Velké multi-agentové simulace a systém NetLogo

NetLogo je široce použitelnou softwarovou platformou pro vytváření mulit-agentových simulací (Wilensky, 1999). Obliba tohoto systému souvisí se jeho specifickými vlastnostmi, mezi které patří:

- rychlost, s jakou lze postupovat od návrhu modelu přes jeho programování až ke spuštění simulace a provádění experimentů,
- přehledná a srozumitelná dokumentace,
- velká knihovna hotových modelů z různých aplikačních domén,
- dostupnost NetLoga – je poskytováno zdarma,
- dostupnost různých rozšíření (extensions) pro NetLogo poskytujících další funkčnosti.

Navzdory těmto kvalitám není NetLogo vnímáno jako dostatečně efektivní systém, který by poskytoval vysoký výkon a výpočetní rychlost pro rozsáhlé modely.

Příkladem opravdu rozsáhlého modelu může být např. model EURACE – masivně multi-agentový model ekonomiky zemí celé Evropské Unie, tento model je popsaný v (Deissenberg at al., 2008). Celkový počet agentů v tomto modelu dosahuje úrovně 100 milionů. Model používá pro běh simulace prostředí pro flexibilní agentové modely velkého rozsahu FLAME (Flexible Large-scale Agent Modelling Environment). Výkonnost platformy FLAME pro model EURACE na čtyřech různých superpočítačích ukazuje graf na obr. 23. Je zde zachycena závislost celkové doby trvání jedné iterace (jednoho simulačního kroku) na počtu procesorů podílejících se na výpočtech. Např. pro 50 procesorů můžeme z grafu předpokládat dobu trvání jedné iterace zhruba 50 sekund, to znamená, že jeden procesor je schopný za jednu sekundu zpracovat výpočty pro 40 000 agentů.



Obrázek 23 - Platforma FLAME, zdroj: (Deissenberg at al., 2008)

Pro srovnání mějme jednoduchý model v NetLogu, který v prostředí o rozměru 200 x 200 buněk pro 40 000 agentů provádí v jednom simulačním kroku pro každého agenta modelový výpočet, který využívá parametry ostatních agentů nacházejících se v blízkém okolí, např. výpočet průměrných [x, y] souřadnic ostatních agentů nacházejících se v kruhovém okolí s poloměrem 10 buněk. V tomto modelu trvá jeden výpočetní krok na standardním PC (CPU Intel i5, 2,30 GHz, 8 GB RAM) přibližně 8 sekund.

Simulace v NetLogu tedy narážejí na hranici počtu agentů, za kterou je průběh simulace natolik pomalý, že výsledky simulace jsou v akceptovatelném čase prakticky nedosažitelné.

V článku (Railsback at al., 2017) se autoři snaží tomuto vnímání omezení NetLoga čelit a navrhují několik technik přinášejících zvýšení rychlosti simulací:

- použití příkazů pro filtrování agentů,
- správné použití globálních a lokálních seskupení agentů (agentsets),
- použití proměnných agentů a prostředí (patchses),
- vyhnutí se opakovaným inicializacím,
- použití stavových proměnných místo spojů (links) mezi agenty,
- použití alternativních příkazů,
- použití rozšíření (extension), např. table.

Tyto techniky se týkají ve většině případů správného použití vlastních NetLogo programovacích prostředků a konstrukcí. V jednom případě návrh zmiňuje použití NetLogo rozšíření (extension). Vytváření rozšíření pomocí dostupného NetLogo API je silnou vlastností systému NetLogo. NetLogo je aplikace napsaná v jazyce Java, rozšíření mohou být též napsána v Javě, případně se mohou opírat o další knihovny či API, včetně využití dalších programovacích jazyků (C++). Ověření tohoto přístupu je popsáno např. v článku (Procházka & Olševičová, 2014), kde je prezentováno řešení pro převod video informace na informace o polohách agentů a jejich zpracování v reálném čase v multi-agentovém systému, je použito programové rozšíření NetLoga o funkčnosti zpracování videa, rozpoznávání objektů a převod video informace na informaci o pohybujících se objektech (chodcích).

Jedním z dílčích cílů dizertační práce, bylo ukázat, jak může NetLogo rozšíření využívající prostředků výpočtů v heterogenních systémech, specificky pomocí platformy OpenCL, rapidně zvýšit rychlost simulací v NetLogu a jiných simulačních systémech postavených na platformě Java, zejména pro masivně multi-agentové simulace.

3.6.4 Paralelizace v multi-agentových systémech

Aplikace využívající obecných paralelních výpočtů v heterogenních výpočetních systémech se vždy skládají ze dvou částí kódu: části určené pro běh na hlavním procesoru systému (CPU) – této části se

říká host aplikace, a části běžící na vlastním zařízení umožňujícím paralelní výpočty (např. na grafické kartě) – tato část je označována jako aplikace zařízení (device application). V práci (Pavlov a Müller 2013) jsou definovány tři různé možnosti, jakým způsobem mohou být host aplikace a aplikace zařízení zkombinovány pro potřeby multi-agentového systému:

- Agenti na host aplikaci (Agents on host – AoH) – tento model se skládá ze třech hlavních komponent: prostředí, agentů a správce úkolů. Agenti komunikují v části host aplikace prostřednictvím prostředí (ne přímo sami mezi sebou). Prostředí poskytuje veškeré informace o stavu systému. Správce úkolů udržuje informace o požadovaných úkolech a výsledcích plnění těchto úkolů. Je zodpovědný za posílání dat do zařízení (GPU) k paralelnímu zpracování a za spouštění paralelně zpracovaných kernelů. Agenti svoje individuální úkoly delegují prostřednictvím správce úkolů na paralelní výpočetní zařízení, kde jsou stejné kernely provedeny nad rozdílnými daty. Po dokončení výpočtu opět správce úkolů poskytne výsledky agentům a to zprostředkovaně, přes prostředí modelu.
- Agenti na aplikaci zařízení (Agents on device – AoD) – v tomto modelu se prostředí modelu, vlastní agenti i správce úkolů nalézají přímo aplikaci zařízení, tedy na paralelně pracujícím výpočetním zařízení (většinou jako globálně dostupná datová struktura). Rolí správce úkolů je v tomto případě synchronizace nezávisle probíhajících vláken a kopírování dat ze sdílené globální paměti do lokální paměti jednotlivých výpočetních vláken (nebo skupin vláken). Po dokončení výpočtu všech vláken jsou výsledky synchronizovány a poskytnuty host aplikaci.
- Hybridní architektura (HA) – kombinace předchozích dvou modelů, prostředí a agenti mají své reprezentace jak v části host aplikace, tak v aplikaci zařízení. Správce úloh se zde stará o synchronizaci mezi jednotlivými částmi.

Některé platformy již delší dobu poskytují nativní podporu pro paralelní zpracování multi-agentových modelů. Jedná se např. o RepastHPC (Collier & North, 2012), D-Mason (Cordasco at al., 2012), Pandora (Angelotti at al., 2001), Flame (Coakley at al., 2012). Tato podpora paralelního zpracování většinou zahrnuje umožnění společných výpočtů na několika fyzických výpočetních uzlech a distribuci agentů na tyto výpočetní uzly. Tyto platformy jsou detailně zkoumány v (Rousset at al., 2014) nebo v (Siegfried, 2014). Z výše uvedených platforem se paralelizací výpočtů přímo na grafických kartách zabývá platforma FLAME ve své verzi FLAME GPU, která využívá pro paralelizaci výpočtů platformu CUDA, použití je tedy vázáno výhradně na grafické karty výrobce NVIDIA. FLAME GPU využívá např. práce (Heywood at al., 2015) pro paralelizaci mikro-simulačního modelu silniční sítě.

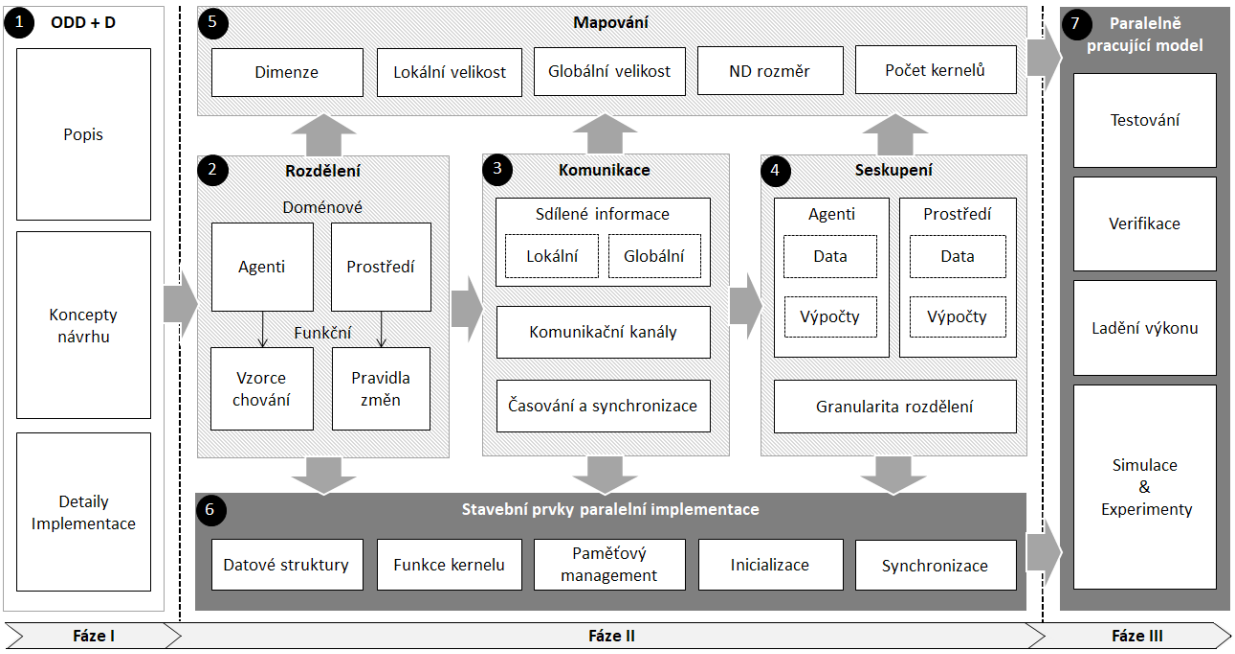
4 Metodika návrhu paralelních multi-agentových modelů

Tato kapitola se věnuje prvnímu disertačnímu dílčímu cíli – vypracování metodického postupu pro návrh a implementaci multi-agentových systémů s paralelním zpracování. Navržený metodický postup se opírá o tři vzájemně propojené pilíře:

- Definice modelu prostřednictvím protokolu ODD+D
- Fosterova metodika návrhu paralelních aplikací
- Principy objektového návrhu OOP a agentově-orientovaného návrhu AOP

Metodický postup se skládá z kroků, které postupně transformují požadavky na model, zachycené prostřednictvím protokolu ODD+D, na stavební prvky paralelní implementace modelu. Tyto implementační části nakonec vytvoří ucelený, paralelně pracující systém, který umožní nad modelem provádět simulace a experimenty. Schematicky je metodický postup znázorněn na obr. 24.

Pro snadnější orientaci je metodický postup rozdělen do tří fází. První fáze pokrývá popis modelu pomocí protokolu ODD+D. Tento popis je vstupem pro druhou fázi, ve které probíhá vlastní návrh paralelního zpracování. Druhou fázi tvoří Fosterovou metodika upravená pro účely paralelizace multi-agentových modelů spolu s několika přidanými kroky, které jsou specifické pro návrh paralelizace pomocí obecných výpočtů na grafických kartách (přizpůsobenými pro platformu OpenCL). Třetí fáze obsahuje kroky, které je třeba provést pro otestování, verifikaci a vyladění výkonu paralelizovaného. Součástí třetí fáze je i návrh provedení simulací a experimentů nad paralelizovaným modelem.



Obrázek 24 – Metodický postup pro návrh a implementaci paralelního multi-agentového modelu, zdroj: Autor

4.1 Popis modelu – ODD+D protokol

Popis modelu pomocí protokolu ODD+D poskytuje velmi přesný vstup pro další kroky metodického postupu návrhu paralelní verze takto popsaného modelu. Na obr. 24 je tento popis označen jako sekce 1. Protokol ODD+D byl popsán v kap. 3.6.1 a jeho detailní struktura je dostupná v příloze A. Tento protokol poskytuje velmi užitečný formát, v jakém je model popsán. Zejména kvůli detailní struktuře zaměřující se na všechny důležité aspekty modelu (např. kompozici modelu, druhy agentů a jejich vnímání či rozhodování, skupinové chování, komunikaci) a zejména kvůli jasně definovaným otázkám, které si tvůrce modelu musí položit, aby model v jednotlivých aspektech správně popsal. Předmětem tohoto metodického postupu není vlastní tvorba popisu modelu v protokolu ODD + D, tímto tématem se zabývají práce jejichž reference jsou uvedeny v kap. 3.6.1. Pro tento metodický postup je podstatné:

- Jaké konkrétní položky protokolu ODD + D jsou důležité pro návrh paralelní verze modelu.
- Jaké typy odpovědí na protokolem předepsané otázky pro tyto položky je možné dostat.
- Jak se tyto odpovědi promítnou do návrhu paralelního modelu.

Další popis metodického postupu v následujících kapitolách je koncipován tak, aby obsahem odpovídal právě trojice: otázky–možné odpovědi–promítnutí odpovědí do návrhu.

4.2 Doménové a funkční rozdělení

Kroky rozdělení se zabývají identifikací potenciálu modelu k paralelnímu zpracování. Hledají odpověď na otázku, co všechno se vlastně v modelu může paralelizovat (zatím se zde záměrně neřeší otázka jak). Na obr. 24 je skupina kroků týkajících se rozdělení označena jako sekce 2. Fosterova metodika navrhuje dva možné přístupy k rozdělení – doménovou a funkční dekompozici. Odpovídajícím předmětem zájmu jsou v multi-agentových modelech agenti a prostředí, ve kterém se agenti pohybují.

Pro doménové rozdělení jsou podstatné následující informace:

Agenti:

- Jaké všechny druhy agentů se v modelu vyskytují?
- Jaké jsou všechny atributy (stavové veličiny) jednotlivých druhů agentů?

Prostředí modelu:

- Jak jsou organizovány buňky prostředí modelu?
- Jaké jsou atributy prostředí jako celku?
- Jaké jsou všechny atributy jednotlivých buněk prostředí?

Odpovědi na tyto otázky poskytují statický pohled na entity modelu. Výstup může být zpracovány do podoby **diagramu tříd**, který obsahuje jen atributy objektů jednotlivých tříd, ne jejich metody.

Pro funkční rozdělení jdou podstatné tyto informace:

Agenti:

- Z jakých vzorců chování a pohybu se skládá jeden simulační krok pro jednoho agenta daného druhu?
- Jaké jsou konkrétní aktivity, které agent musí vykonat pro realizaci předepsaných vzorců chování nebo pohybu?
- Jaká je výpočetní realizace těchto aktivit?

Prostředí:

- Jak se mění prostředí modelu v rámci provedení jednoho simulačního kroku?
- Jak se mění během simulačního kroku jednotlivé buňky prostředí?

Vztah agentů a prostředí:

- Jak se agenti jednotlivých druhů po prostředí pohybují? Diskrétně po jednotlivých buňkách nebo spojitě po jakýchkoli souřadnicích?

Odpovědi na tyto otázky poskytují dynamický pohled na model. Do výčtu aktivit je třeba zahrnout všechny aktivity spojené s vnímáním, rozhodováním a vykonáváním vzorců chování a pohybu agentů stejně jako všechny změnové operace, které se dějí v prostředí i jednotlivých buňkách. Je třeba specifikovat velmi konkrétní matematické postupy těchto aktivit a změn.

Výstup funkčního rozdělení může být zachycen nejlépe pomocí **diagramů aktivit** pro jednotlivé třídy objektů (typy agentů), pro prostředí a pro jednotlivé buňky. A také pomocí **sekvenčních diagramů**, které aktivity jednotlivých tříd objektů dávají do časových souvislostí. Tento diagram aktivit je třeba doplnit o detaily, jak jsou jednotlivé aktivity realizovány jako skutečný matematický výpočet. Pro funkční dekompozici je mnohem podstatnější skladba prováděních aktivit a způsob jejich skutečné realizace než fakt, že těmito schopnostmi disponuje konkrétní typ agenta nebo buňka prostředí.

Výstupem rozdělení jsou:

- Seznam atributů agentů a buněk, které se během simulačního kroku mění (měnící se atributy mohou být vyznačeny v diagramu tříd).
- Výčet všech prováděných výpočtů pro realizaci aktivit agentů (jejich pravidel chování a pohybu) a pravidel změn prostředí a jednotlivých buněk. Jde o detaily k jednotlivým aktivitám.

Na základě těchto souhrnů je možné identifikovat jaké výpočty pro realizaci aktivit agentů či změn prostředí jsou prováděny opakovaně a nezávisle a mohou tak být předmětem paralelního zpracování.

4.3 Komunikace

Předchozí sekce se zabývala agenty a buňkami prostředí jako individuálními entitami a jejich soukromými výpočty. Kroky ze sekce komunikace (na obr. 24 označené jako č. 3) oproti tomu hledají styčná místa mezi těmito entitami, ve kterých musí dojít k výměně informací o stavových veličinách ostatních agentů či buněk či mezivýsledcích připravených jinými entitami. Pro realizaci komunikace při paralelním zpracování je třeba nalézt odpověď na následující otázky, a to jak pro agenty, tak pro buňky prostředí (nebo prostředí jako celek):

- Jaké externí informace agent či buňka potřebují pro realizaci svých vlastních, vnitřních (individuálních) výpočtů?
- Kdo tyto informace produkuje (počítá) nebo vlastní (hodnoty atributů agentů či buněk) a kdo je konzumuje?
- Jsou tyto informace sdíleny jen mezi některými agenty/buňkami nebo jdou dostupné globálně pro všechny?
- Kdy přesně jsou, z pohledu sekvence průběhu výpočtu, sílené informace produkovány, kdy musí dojít k jejich sdílení (tj. kdy přesně jsou spotřebovávány cílovými agenty nebo buňkami)?

Na základě odpovědí na tyto otázky je možné navrhnout:

- Jakými komunikačními kanály budou tyto informace sdíleny (pomocí posílání zpráv nebo pomocí sdílených datových struktur).
- Jaké konkrétní sdílené datové struktury je nutné použít.
- V jakých místech je třeba komunikaci synchronizovat tak, aby všechny individuální entity měly pro své vlastní výpočty včas všechny potřebné vstupy od ostatních entit.

Komunikační potřeby je možné zobrazit jako **diagram komunikací** (takto nazvaný v UML 2, v UML 1 se tento diagram nazývá diagram spolupráce) tento diagram říká kdo, s kým, co a jak komunikuje. Diagram obsahuje:

- **Uzly** – objekty agentů nebo buněk, které hrají roli odesílatele a vysílače v Shannon-Weaverově modelu komunikace (odesílatel-vysílač-kanál-přijímač-adresát).
- **Komunikační cesty** – hrají roli kanálu pro přenos informace.
- **Zprávy** – konkrétní informace, které jsou komunikovány. Jednotlivé identifikované komunikační potřeby budou odpovídat konkrétním typům komunikačních zpráv (např. nová x-

ová a y-ová souřadnice jednoho konkrétního agenta komunikovaná okolním agentům, aby měli informaci o nové poloze dotyčného agenta).

- **Sekvenční výrazy** - definují pořadí, ve kterém jsou jednotlivé zprávy (informace), které využívají konkrétní komunikační cestu posílány.

Pro zachycení časování (vzniku, přenosu spotřeby sdílených informací) je možné využít **sekvenčního diagramu**. Ten zachycuje časové souvislosti výměny informací mezi jednotlivými entitami (agenty nebo buňkami).

Při paralelním zpracování výpočtů v heterogenních systémech, speciálně pokud jsou realizovány jako obecné výpočty na grafických kartách, jako komunikační kanál slouží vhodná sdílená datová struktura. Poslat zprávu potom znamená zaznamenat informaci svým původcem-odesílatelem (výpočetním vláknem reprezentujícím agenta či buňku) do této datové struktury. Datová struktura je sama o sobě komunikačním kanálem a příjemce zprávy (adresát) si sdílenou informaci vyzvedne jejím přečtením. Je jasné, že příjemci zprávy tak mohou být všechna výpočetní vlákna, která mají k příslušné datové struktuře přístup. Taková datová struktura musí poskytovat možnost konkurenční přístupu pro čtení i zápis a musí být navržena tak, aby nedošlo k zablokování paralelně běžících vláken, které tuto datovou strukturu využívají.

Více o neblokujících datových strukturách a synchronizaci viz kap. 3.2.2 a dále viz kap. 5.3.2 pojednávající o konkrétní datové struktuře pro komunikaci informací o vzájemné poloze agentů v prostředí modelu.

4.4 Seskupení

Během doménového a funkčního rozložení došlo k identifikaci co nejmenších stavebních prvků – entit, ze kterých se model skládá a k identifikaci elementárních výpočtů, které tyto entity musejí provádět, aby realizovali svoje předepsané vzorce chování a pohybu. Rozložení na elementární prvky umožnilo určit potenciál modelu k paralelizaci. Paralelní zpracování může probíhat přímo nad těmito elementárními entitami. Někdy je ale efektivnější, pokud jsou paralelně zpracovány až některá seskupení elementárních entit. Obecně pro paralelní zpracování platí, že efektivní návrh je ten, ve kterém se „více počítá, než komunikuje“. Požadavky na komunikaci jsou proto hlavním faktorem určujícím, jak intenzivně je třeba seskupením se zabývat.

Kromě ODD+D protokolu jsou užitečným vstupem pro seskupení také informace o organizačním paradigmatu, které bude model používat (viz kap. 3.6.2). Způsob, jakým mají být agenti organizováni indikuje, jak mohou tato uskupení elementárních entit vypadat a jaké jsou požadavky na specializované agenty. Toto jsou důležité otázky pro seskupení:

- Existují skupiny entit, uvnitř kterých probíhá intenzivnější komunikace, než je komunikace se zbylými entitami?
- Mají některé skupiny agentů shodné či velmi podobné cíle?
- Opakují některé entity (agenti nebo buňky) stejné individuální výpočty nebo výpočty nad stejnými daty?
- Dají se některé individuální, elementární výpočty agentů či buněk, ke kterým je zapotřebí externích dat od jiných agentů/buněk, delegovat na entitu vyšší úrovně, která by potřebná data vlastnila?

Odpovědi na tyto otázky poskytnou vodítka, jak se seskupení do návrhu paralelní verze modelu promítne:

- Vytvořením entit zastupujících více agentů a jejich funkčnost.
- Vytvořením komunikačních mediátorů – specializovaných agentů vlastnících informace od jiných agentů a zprostředkujících je ostatním nebo pro ostatní agenty či skupiny agentů zprostředkující rozhodnutí vyšší hierarchické úrovně, než jsou individuální rozhodnutí jednotlivých agentů.
- Vytvořením výpočetních mediátorů – agentů, kteří pro skupiny agentů provádějí výpočty, ke kterým je zapotřebí vlastnit sdílená data, která individuální agenti nemají k dispozici.
- Zvyšováním lokality řešeného problému – omezením komunikačních požadavků od individuálních agentů jejich začleněním do vhodně lokalizované skupiny (např. agenti v určitém prostorovém segmentu prostředí, komunikace poté probíhá mezi segmenty, ne mezi individuálními agenty).

Příkladem může být model, ve kterém bude použito organizační paradigma holarchie. V takovém modelu existují malé, skupiny agentů s podobnými vlastnostmi nebo sledující podobné cíle – holony. Agenti uvnitř skupin spolu potřebují intenzivně komunikovat, navenek jsou ale holony autonomní a jejich komunikaci s ostatními holony (ostatními skupinkami agentů) zprostředkují specializovaní agenti – mediátoři. Pro takový model by bylo efektivnější než paralelní zpracování na úrovni individuálních agentů

4.5 Mapování

Fosterova metodika návrhu paralelních aplikací pojímá fázi mapování jako proces, při které je rozhodnuto, jakým způsobem se jednotlivé elementární výpočetní úlohy budou rozdělovat mezi dostupná výpočetní jádra tak, aby bylo maximalizováno jejich využití a minimalizovány náklady na komunikaci. Toto je v případě použití platformy OpenCL řízeno samotnou platformou. Nicméně stále je třeba definovat dimenze a rozměry úlohy prostřednictvím parametrů které jsou poskytnuty při

spouštění jednotlivých kernelů. Na těchto parametrech závisí, jakým způsobem budou jednotlivá paralelně pracující vlákna organizována.

Mapování je na obr. 24 zobrazeno v sekci 5. Nastavení parametrů rozměrů kernely zpracovávaných úloh je třeba při návrhu paralelně pracujícího modelu zvážit. Toto jsou otázky, které jsou pro volbu parametrů rozměrů úlohy důležité:

- Jaká je povaha paralelní úlohy řešené modelem, pokud jde o její rozměr? (Dimenze úlohy)
- Jaké jsou možnosti zařízení, na kterých bude paralelní výpočet probíhat?
- Jaká je velikost skupin agentů, buněk, kterým budou odpovídat pracovní skupiny paralelně běžících vláken? (Lokální velikost)
- Jaký je celkový rozsah úlohy a čím je určen? (Globální velikost)
- Jak jsou jednotlivé výpočetní úlohy modelu mapovány na funkce kernelu? Proběhne jeden simulační krok v rámci volání jedné funkce kernelu nebo v oddělených, po sobě jdoucích kernelech? (Počet funkcí kernelů)

Nalezení odpovědí na tyto otázky vede k volbě správných parametrů rozsahu paralelní úlohy a k nastavení parametrů kernelu.

4.6 Stavební prvky paralelní implementace

Z předchozích kroků metodického postupu vyplývá, jaké budou konstrukční prvky paralelizovaného modelu. Způsob, jakým budou tyto stavební prvky vytvořeny a propojeny ovlivní jednak efektivitu paralelního zpracování výpočtů v modelu a také způsob, jakým se bude dále s modelem zacházet (např. jak se budou provádět simulace a experimenty). Při návrhu je třeba najít odpověď na následující otázky, které se promítnou do způsobu implementace:

Datové struktury:

- Jaké budou v modelu použity datové struktury pro uchování dat z atributů agentů a z buněk?
- Jak budou uchována data, která nejsou vázána na atributy? Jde např. o pole hodnot, se kterými model počítá, monitorovací proměnné, výstupy výpočtů, které nejsou vázány na atributy agentů či buněk.

Funkce kernelů:

- Jak bude realizace paralelních výpočtů skutečně provedena jednotlivými funkcemi kernelu? Kód funkcí kernelu je třeba naprogramovat v jazyce OpenCL.

- Jak lze využít již naprogramované části kódu, které implementují typické výpočetní bloky, reprezentující výpočty pro určitý druh chování agentů či pravidel změn buněk? (Detaily k znovupoužití částí kódu modelu a kernelů viz implementace paralelních modelů v kap. 6
- V jakém pořadí budou funkce kernelu volány a jak budou synchronizovány? Jedna funkce kernelu reprezentuje výpočet jednoho individuálního vlákna. Je třeba rozhodnout, zda bude simulační krok reprezentován jedinou funkcí kernelu nebo bude třeba volat více kernelů a jejich provedení nějak synchronizovat.

Paměťový management:

- Kdy a jakým způsobem budou v modelu volány funkce pro vytvoření paměťových objektů z atributů agentů nebo buněk? Jaké funkce NL2OCL se pro to použijí? Detaily viz popis funkcí paměťového managementu programového rozšíření NL2OCL v kap. 5.5.
- Kdy a jakým způsobem budou tyto paměťové objekty předány funkcím kernelů jako jejich argumenty? Detaily viz popis funkcí paměťového managementu programového rozšíření NL2OCL v kap. 5.3.1.

Inicializace:

- Jak probíhá inicializace simulace před jejím zahájením?
- Kdy je třeba načíst iniciální data do paměťových objektů? Mohou se iniciální parametry simulace během jejího běhu měnit?

Synchronizace NetLogo–NL2OCL–OpenCL:

- Které paměťové objekty, pro jaké atributy agentů a buněk jsou jen vstupními daty pro výpočty bez zpětné synchronizace do modelu?
- Které atributy agentů a buněk je naopak potřeba synchronizovat po dokončení výpočtů funkcemi kernelu?
- Je třeba synchronizovat vypočítané hodnoty s hodnotami atributů agentů a buněk po každém simulačním kroku nebo jen v nějakých časových okamžicích simulace?

Odpovědi na výše uvedené otázky při návrhu paralelně pracujícího modelu poskytnou vodítko pro rozhodnutí, jaké konkrétní funkce programového rozšíření NL2OCL se použijí, kdy a jakým způsobem.

4.7 Paralelně pracující model

Konečná fáze metodiky návrhu paralelně pracujících modelů se skládá z kroků, které se týkají vlastního použití modelu pro provádění simulací a experimentů – obr. 24, sekce 7. Pro správné použití modelu

je třeba model otestovat a verifikovat jeho chování. Pro dosažení žádoucích efektů paralelizace (zrychlení simulace) je třeba model vyladit.

Testování – testování paralelně pracujících aplikací je obecně komplikovanější, než je tomu při testování aplikací se sekvenčním zpracováním. NL2OCL disponuje funkcionalitou, která umožňuje snadno přidat do modelu prvky, které testování usnadní. Jednak jde o monitorovací proměnné, které se do modelu dají snadno zavést ve formě separátních paměťových objektů. Tyto objekty nejsou vázány na žádné atributy agentů nebo jsou navázány na atributy agentů vytvořené speciálně pro testovací účely. S těmito objekty je následně možné uvnitř kernelů manipulovat, sbírat do nich informace o průběhu výpočtu a následně je synchronizovat zpět do prostředí modelu a tam je zobrazit. Už při návrhu modelu je možné s těmito testovacími proměnnými počítat, zejména pro použití v klíčových místech paralelního výpočtu, v místech rozhodování agentů, v bodech synchronizace apod.

Verifikace – verifikace paralelně pracujícího modelu spočívá v porovnání chování při běhu simulace se standardní verzí modelu. Obě varianty by měly poskytovat stejné výsledky. Srovnáním běhu paralelní a standardní simulace by mělo být provedeno se stejnými iniciálními parametry, rozložením agentů a stavem prostředí.

Ladění výkonu – po implementaci paralelně pracujícího modelu by mělo dojít k ověření dosaženého zrychlení. Toto ověření lze provést opakovanými simulacemi s měnícími se parametry rozsahu a náročnosti simulační úlohy (viz vybrané ukázkové paralelizované modely - kap. 6). Profil dosažených zrychlení umožní analyzovat škálovatelnost paralelní implementace (žádoucí je, aby zrychlení při zvětšujícím se rozsahu simulační úlohy také rostlo). Pokud je paralelní implementace špatně škálovatelná nebo skutečné zrychlení nedosahuje zrychlení předpokládaného, je třeba ověřit především tyto specifické komponenty paralelního návrhu:

- Realizace požadavků na komunikaci – přenosy dat mezi modelem NL2OCL a OpenCL jsou časově náročné v porovnání s vlastními paralelními výpočty. Je třeba provést návrh tak, aby většina výpočetního času byla věnována vlastním výpočtům a zanedbatelná část komunikaci a paměťovým přesunům.
- Synchronizační mechanizmy mohou výpočet značně ovlivnit, zejména neefektivní použití globálních synchronizačních prostředků v kernelech (kdy se v kernelu na určitém místě v kódu čeká, až se na toto místo dostanou všechna spuštěná vlákna).
- Nastavení parametrů rozsahu úlohy. Je třeba přizpůsobit tyto parametry možnostem konkrétního zařízení, na kterém jsou výpočty paralelně zpracovány. Je dobré konzultovat nastavení se specifikací dodavatele HW. Každé zařízení má danou optimální velikosti lokálních pracovních skupin paralelně běžících vláken a určité možnosti dimenzí úlohy.

Simulace a experimenty – už při návrhu paralelně pracujícího modelu by mělo být zváženo, jaké konkrétní simulace a experimenty budou s modelem prováděny. Pro vlastní implementaci modelu jsou důležité z pohledu plánovaných simulací a experimentů zejména tyto aspekty:

- Jakým způsobem bude model parametrizován na začátku a v průběhu simulace?

Každá změna parametrů simulace za běhu se musí propagovat z modelu přes NL2OCL do OpenCL kernelu, který provádí paralelní výpočty. To může znamenat zpomalení paralelního zpracování. Lze to řešit například synchronizací parametrů běhu modelu jen v určitých jasně definovaných okamžicích, např. jednorázově na základě aktivity uživatele modelu.

- Jaké simulační výstupy a jakým způsobem budou prezentovány uživateli?

Jde o to, jak bude uživatelské rozhraní modelu synchronizováno s vlastním výpočtem. Aktualizace grafické reprezentace modelu může být na čas velmi náročnou operací v porovnání s vlastním výpočtem simulačního kroku (zejména v tzv. mikro-simulacích, které pokrývají velmi detailní vzorce rozhodování a chování agentů). Při návrhu paralelně pracujícího modelu by mělo být jasné, kdy se bude grafická reprezentace modelu aktualizovat a zda to nebude mít negativní dopad na rychlost simulace. Negativní dopady aktualizace grafické reprezentace stavu simulace lze eliminovat podobně jako u parametrizace modelu prováděním aktualizace jen v jasně specifikovaných časových okamžicích, např. vždy po provedení určitého počtu provedených simulačních kroků.

- Jaká data budou sbírána v průběhu simulace?

Pro návrh experimentů je důležité specifikovat, jaká data se budou v průběhu simulace sbírat a jakým způsobem se budou zaznamenávat. Toto může ovlivnit implementaci funkcí kernelů, pokud se sbíraná data generují právě v nich. Sběr dat v modelu mezi simulačními kroky, tj. mezi jednotlivými voláními kernelů, může také ovlivnit rychlost simulace. Podobně jako nastavování parametrů či aktualizace grafické informace může být řešení zvýšení poměru provedených simulačních kroků (tj. vlastních paralelních výpočtů) k počtu okamžiků, ve kterých dochází ke sběru dat. Další možností je sběr dat prostřednictvím speciálních paměťových objektů, které budou plněny přímo v kernelech a jejichž hodnoty budou vyzvednuty až na konci simulace – takové řešení nebude mít na rychlost výpočtu prakticky žádný negativní dopad.

5 Programové rozšíření NL2OCL

5.1 Vytvoření NL2OCL – softwarový projekt

Softwarové řešení NL2OCL je hlavním technickým výstupem této dizertační práce. Jedná se o programové rozšíření pro multi-agentový systém NetLogo umožňující přímo z modelu využívat rychlé výpočty na grafických kartách. Verze prezentovaná v této práci, označená jako verze 1.0, je připravená pro distribuci a použití dalšími tvůrci modelů. Informace v této kapitole pokrývají implementační a dílem i uživatelskou dokumentaci, navíc jsou všechny zdrojové kódy opatřeny rozsáhlou dokumentací ve formátu JavaDoc. Prezentace převedení vybraných modelů na verze s paralelními výpočty, kterou se zabývá celá kapitola 6 může sloužit jako průvodce, jak NL2OCL použít na konkrétních modelech.

Práce na NL2OCL programovém rozšíření byla pojata jako softwarový projekt realizovaný pomocí agilního přístupu inspirovaného hlavně metodickým rámcem Scrum. To znamená, že cyklus vývoje od sběru požadavků, přes návrh řešení a implementaci až po vlastní nasazení a testování s konkrétními modely byl velmi krátký, opakoval se cca v týdenních intervalech. Celý vývoj byl řízen primárně požadavkem na robustní funkčnost a snadnost použití NL2OCL uživateli, kteří nemají a nepotřebují rozsáhlé zkušenosti s platformou OpenCL a s technickými detaily výpočtů na grafických kartách. Snahou bylo prostředky OpenCL co nejvíce zapouzdřit a odstínit od vlastních multi-agentových modelů. Koncepce a technické řešení použité v produktu NL2OCL, primárně určeném pro systém NetLogo, jednoduše použitelné i pro jiné multi-agentové systémy, zejména ty, které jsou postaveny na platformě Java.

Práce na NL2OCL lze rozdělit na dvě hlavní fáze:

- Studie proveditelnosti a vytvoření prototypu – Průzkum technických možností propojení platforem OpenCL a NetLogo a ověření potenciálu zrychlit práci výpočetně náročných modelů. Cílem této fáze bylo vytvoření funkčního prototypu a jeho vyzkoušení na demonstračním modelu.
- Rozšíření a příprava hotového produktu – Na základě prototypu a požadavků pramenících z praktického použití byla funkčnost NL2OCL podstatně rozšířena. Některé části byly re-designovány kvůli zvýšení celkové robustnosti řešení a lepší uživatelské použitelnosti.

5.2 Analýza požadavků

Sběr funkčních požadavků na NL2OCL probíhal kontinuálně na základě aktuálně řešených technických problémech. Sále existují další požadavky, které jsou připraveny k zapracování a jistě se bude NL2OCL dále rozvíjet. Přesto se již v ranné fázi prací na NL2OCL vyprofilovaly klíčové požadavky, jejichž

implementace byla nezbytná pro úplný běh OpenCL aplikace z prostředí NetLoga. K implementaci těchto klíčových požadavků došlo již v prototypu NL2OCL.

Následující přehled uvádí funkční požadavky rozdělené do třech kategorií:

I. Konfigurace prostředí běhu aplikace
   Tato kategorie požadavků se týká nastavení prostředí běhu OpenCL aplikace, tento úkol vyžaduje provedení několika kroků
II. Management paměti
   Tato oblast pokrývá manipulaci s datovými strukturami, výměnu dat mezi agentovým systémem a vlastní OpenCL aplikací stejně jako výměnu dat mezi OpenCL aplikací a OpenCL zařízením (grafickou kartou). Požadavky musejí pokrýt celý cyklus explicitního managementu paměti, který je v OpenCL vyžadován.
III. Správa kernelů
   Vlastní vykonání paralelních výpočtů je prováděno kernelem, tato kategorie požadavků je zaměřena na přípravu kernelů, zpracování zdrojových kódů kernelů, nastavení argumentů funkcím kernelů týkajících se rozměrů úlohy a vlastních dat se kterými kernel počítá. Kernely je třeba spouštět a monitorovat jejich provedení.

Tabulka 4 uvádí funkčnosti, které byly definovány jako **nezbytné** pro ucelenou funkčnost NL2OCL, tyto požadavky byly zapracovány jako první, pokrýval je již prototyp NL2OCL:

Tabulka 4 – Funkční požadavky na programové rozšíření NL2OCL označené jako nezbytné, zdroj: Autor

Funkční oblast	Požadavek	#
Konfigurace prostředí běhu aplikace	Konfigurace a správa zdrojů potřebných pro běh OpenCL aplikace.	I.1
	Poskytnutí informací o dostupných platformách a zařízeních.	I.2
	Vytvoření kontextu běhu programu OpenCL pro specifikovanou platformu a OpenCL zařízení.	I.3
	Vytvoření fronty úkolů pro vybranou platformu a zařízení.	I.4
Management paměti	Vytváření paměťových bufferů pro použití OpenCL zařízením.	II.1
	Propojení datových struktur agendového modelu (atributů agentů a prostředí) a datovými strukturami OpenCL.	II.2
	Načtení dat z modelu do datových struktur OpenCL, tj. plnění paměťových objektů hodnotami z atributů agentů nebo buněk (polí).	II.3
	Zajištění transferů mezi paměťovými objekty OpenCL host aplikace a globální pamětí OpenCL zařízení.	II.4

	Zpětné nastavení hodnot atributů agentů a prostředí modelu na základě hodnot uložených v datových strukturách OpenCL.	II.5
	Správná alokace a uvolňování použité paměti (tento požadavek se stává ještě důležitějším při velkém rozsahu simulací a s tím spojeném velkém rozsahu dat přenášených mezi NetLogo modelem, OpenCL aplikací a OpenCL zařízením.	II.6
Správa kernelů	Příprava kernelu z textu – kompilace zdrojové programu obsahujícího kernel v jazyce C – OpenCL z kódu zadaného přímo v modelu jako řetězec.	III.1
	Sestavení zkompilovaných kernelů a jejich příprava na spuštění OpenCL zařízením.	III.2
	Nastavování argumentů různých datových typů do připravených kernelů	III.3
	Nastavení rozsahu řešeného problému pro kernel, tj. dimenzí úlohy (NDRange), globálního a lokálního počtu pracovních jednotek (Wokr Item).	III.4
	Vlastní vykonání kernelu a monitoring výsledku vykonání.	III.5
	Vyzvednutí vypočítaných výsledků – přenos vypočítaných výsledků z OpenCL kernelů zpět do OpenCL host aplikace	III.6

Požadavky v tabulce 5, byly identifikovány jako **důležité**. Přispívají k robustnosti řešení a snadnosti použití – tyto požadavky byly již také do stávající verze NL2OCL zapracovány:

Tabulka 5 - Funkční požadavky na programové rozšíření NL2OCL označené jako důležité, zdroj: Autor

Funkční oblast	Požadavek	#
Konfigurace prostředí běhu aplikace	Poskytnutí jednoduchého NL2OCL příkazu, který provede celou sekvence kroků k přípravě prostředí běhu OpenCL automatizovaně.	I.5
	Jednoduchá inicializace/uvolnění všech OpenCL zdrojů přímo z modelu pro snadné opakování simulace se znovu-iniciovaným OpenCL prostředím.	I.6
	Zprostředkování informací o návratových hodnotách a chybových hlášení interních funkcí platformy OpenCL do prostředí modelu v NetLogo.	I.7
Management paměti	Inicializace paměťového bufferu na základě agent setu – vytvoření OpenCL paměťového bufferu naplněného hodnotami konkrétního atributu agentů z nějakého specifikované množiny agentů (agent setu).	II.7

	Inicializace paměťového bufferu na základě druhu agenta (množinu agentů tvoří všichni agenti daného typu – breed)	II.8
	Automatizace datových přesunů mezi modelem, OpenCL aplikací a OpenCL zařízením – uživatel má možnost označit při inicializaci paměťových bufferů, pro které atributy bude prováděna automatická synchronizace po dokončení výpočtu kernelem.	II.9
	Poskytnutí speciálních datové struktury pole obsazenosti buněk pro sdílení informací o poloze ostatních agentů při paralelním zpracování výpočtu – datová struktura PA (Patch-Agent).	II.10
	Vytvoření prostředků monitoringu obsahu datových struktur pro snadnější ladění modelu.	II.11
Správa kernelů	Příprava kernelů ze souboru – v modelu je specifikováno pouze jméno souboru, o zbytek se postará NL2OCL.	III.7
	Práce s více funkcemi kernelů definovanými v jednom zdrojovém souboru, automatické rozeznávání jmen funkcí kernelů.	III.8
	Poskytnutí detailních informací o průběhu kompilace a sestavování kernelů – dostupnost chybových hlášení a varování OpenCL kompilátoru.	III.9

5.3 Implementace NL2OCL

Implementace NL2OCL se skládá z několika propojených komponent:

- Java část – implementace NetLogo rozšíření (NL2OCL.jar)

  Tato část se skládá jednak ze standardní implementace NetLogo rozšíření využívající NetLogo API používající jednu instanci třídy org.nlogo.api.DefaultClassManager pro řízení volání funkcí rozšíření a dále sadu instancí třídy org.nlogo.api.Reporter, které implementují vlastní funkce rozšíření.
- C++ část – rychlé výpočty pomocí OpenCL (NL2OCL.dll)

  Tato dynamická knihovna tříd představuje zapouzdření funkcí platformy OpenCL. Poskytuje API pro volání funkcí, které provádějí konfiguraci prostředí běhu OpenCL aplikace, dále poskytuje funkce paměťového managementu, a nakonec funkce pro přípravu kernelů, nastavování jejich argumentů a spouštění.
- JNI část – rozhraní mezi JAVA a C++ částí

Rozhraní pro volání funkcí naprogramovaných v C++ z Java části NL2OCL rozšíření a dostupných v rámci dynamické knihovny tříd je implementováno jako singleton, jehož instance se používá v Reporterech.

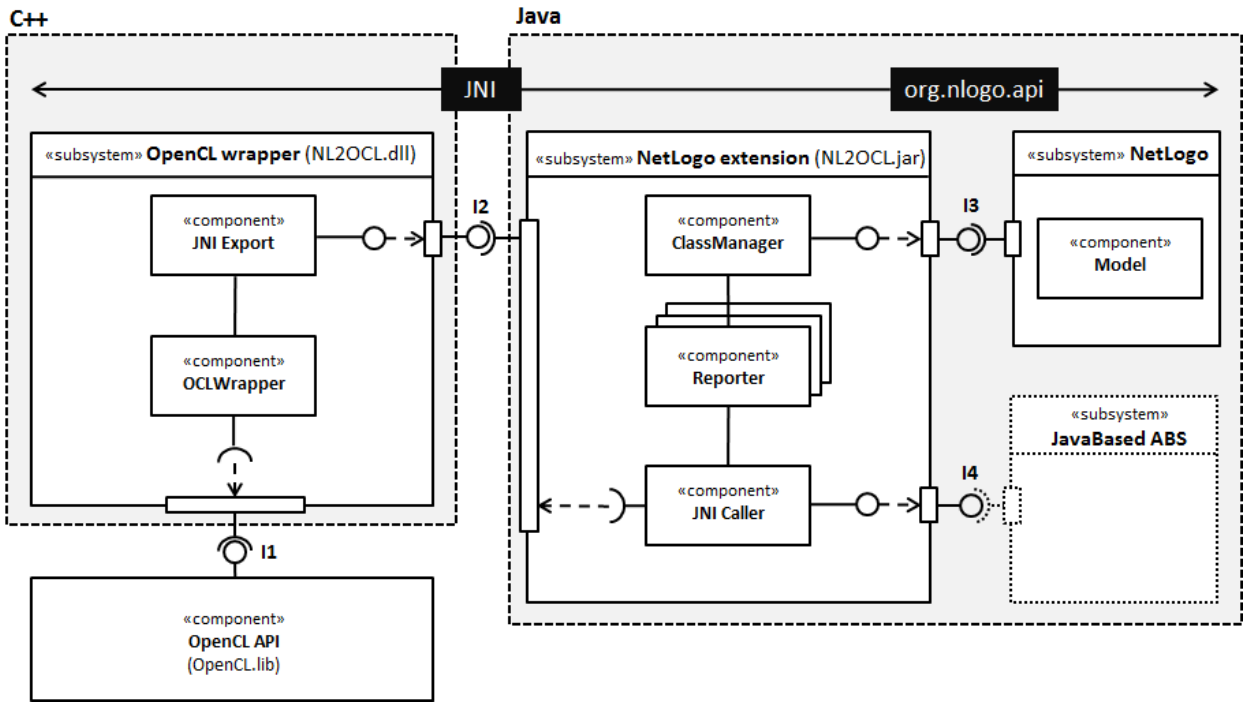
Propojení mezi jednotlivými komponentami zabezpečují rozhraní, viz obr. 25:

I1 – propojení mezi OpenCL knihovnami a dynamickou knihovnou funkcí,

I2 – JNI interface mezi Java a C++ částí implementace,

I3 – propojení modelu v NetLogu s rozšířením NL2OCL,

I4 – potenciální propojení jiného, na Javě postaveného agentového systému (např. AnyLogic), s funkcionalitou NL2OCL, v takovém případě není třeba využívat ClassManager a Reportéry.



Obrázek 25 - NL2OCL: diagram komponent, zdroj: Autor

Při vývoji programového rozšíření NL2OCL byly použity čtyři odlišné technologie. Jim odpovídající čtyři odlišné programovací jazyky – nlogo, Java, C++ a C-OpenCL. A těmto jazykům odpovídá souběžné použití čtyř vývojových prostředí. Z tohoto pohledu je možné považovat vývoj NL2OCL za netradiční.

Tato kombinace vývojových prostředí je samozřejmě pouze jednou z možností. Nicméně tato kombinace jednak splňuje požadavek, že všechny prvky jsou volně dostupné a představuje vyzkoušený, opakovatelný postup pro vývoj programových rozšíření pro systém NetLogo nebo jiné multi-agentové systémy na platformě Java za použití knihoven naprogramovaných v C++. Použité technologie a vývojová prostředí jsou shrnuty v tabulce 6.

Tabulka 6 - použité technologie a vývojová prostředí při vytváření programového rozšíření NL2OCL, zdroj: Autor

Použitá technologie	Vývojové prostředí	Verze
nlogo	Multi-agentový systém NetLogo	6.0
Java	IDE – Eclipse Oxygen	4.7
C++	IDE – Microsoft Visual Studio Community 2015	14.0
C-OpenCL	Notepad++	7.5.1

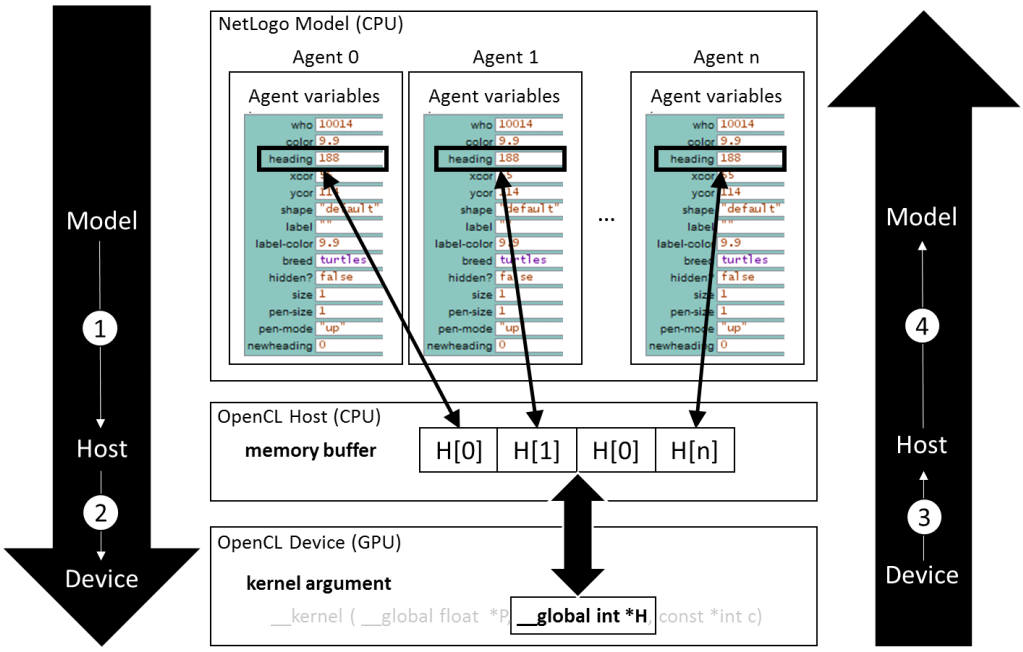
Význam jednotlivých částí je následující:

- Java projekt
    - Obsahuje třídu JNICaller, ve které jsou deklarovány prototypy nativních funkcí, které implementuje až C++ část, která zapouzdřuje OpenCL funkcionalitu. Na základě třídy JNICaller je vyprodukován hlavičkový soubor nl2ocl_jni_JNICaller.h
    - Výstupem Java projektu je Java archiv NL2OCL.jar --> ten je umístěn do specifického adresáře určeného pro programová rozšíření NetLoga: NETLOGO_HOME\app\extensions\NL2OCL\NL2OCL.jar
- C++ projekt
    - Implementuje nativní funkce deklarované v Java třídě JNICaller, prototypy C++ funkcí jsou v hlavičkovém souboru nl2ocl_jni_JNICaller.h
    - Poskytuje vlastní funkcionalitu OpenCL platformy.
    - Produkuje dynamickou knihovnu NL2OCL.dll --> ta se umístí do adresáře: NETLOGO_HOME\app\NL2OCL.dll
- NetLogo model
    - Používá programové rozšíření NL2OCL pomocí příkazu extensions[NL2OCL]
    - Volá jednotlivé funkce NL2OCL:<název funkce> (každá taková funkce je buďto Reporter – výsledek funkce musí být přiřazen, protože funkce má nějakou návratovou hodnotu (objekt), nebo Command – bez návratové hodnoty, jen se provede).
- C-OpenCL kernel
    - Zdrojový kód funkcí kernelu je napsán v jazyce C-OpenCL, specifické podmnožině jazyka C.
    - Zdrojový kód se využije ve funkci NL2OCL:PrepareKernels <název souboru>
    - Výstupem je zkompilovaný a sestavený program pro konkrétní OpenCL zařízení (zařízení, které bylo specifikováno při vytváření OpenCL aplikačního kontextu).

5.3.1 Explicitní paměťový management v NL2OCL

Pomocí NL2OCL funkcí pro práci s paměťovými objekty je možné plnit požadavky OpenCL na explicitní management paměti, který je realizován čtyřmi typy paměťových přesunů. Paměťové přesuny jsou zobrazeny na obr. 26:

1. NetLogo -> Host – kopírování hodnot z atributů agentů (či polí) do paměťového bufferu host OpenCL aplikace (např. pomocí funkce: NL2OCL:CreateFloatBufferFromAgents).
2. Host –> Device – přenos paměťového objektu z OpenCL host aplikace do device části aplikace. Aby mohla grafická karta s daty pracovat, musí být data nejprve přenesena z host aplikace do globální paměti zařízení. Tento přenos probíhá v rámci nastavení argumentů připraveného kernelu (např. pomocí funkce: NL2OCL: SetKernelBufferArgument).
3. Device -> Host – vyzvednutí výsledků z globální paměti zařízení (spočítaných kernelem) do paměťového objektu host aplikace (např. pomocí funkce: NL2OCL:GetFloatBuffer).
4. Host -> NetLogo – přenos výsledků OpenCL aplikace zpět do NetLoga, nastavení hodnot do vybraného parametru agentů/polí (např. pomocí funkce: NL2OCL:CopyFloatBufferToAgents).



Obrázek 26 - Paměťové transfery v NL2OCL, zdroj: Autor

Pokud jsou správně nakonfigurovány platforma a OpenCL zařízení, připraven OpenCL aplikační kontext a fronta úkolů (pomocí funkce NL2OCL:PrepareEnvironment), následná práce s OpenCL kernely je potom realizována v několika krocích:

1. příprava kernelu ke spuštění pomocí funkce NL2OCL:PrepareKernel,

2. nastavení argumentů kernelu pomocí funkce NL2OCL:SetKernelArgument (pro daný datový typ argumentu),
3. spuštění kernelu,
4. vyzvednutí výsledků spočítaných kernelem.

Spouštění a vyzvedávání vypočítaných výsledků kernelem může probíhat pro jeden kernel opakovaně, není třeba znovu kernel připravovat (překlad a sestavení) a dodávat mu vstupní argumenty. V tom spočívá výpočetní síla tohoto řešení – kernel se připraví jednou, dodá se mu sada vstupních argumentů a po každém spuštění kernelu se pouze vyzvednou spočítané výsledky. Kernel pak může být okamžitě znovu spuštěn, přičemž aktuálními vstupními argumenty kernelu se stávají právě spočítané hodnoty. Po každém spuštění kernelu jsou vypočítané výsledky buďto k dispozici k vyzvednutí anebo může výpočet pokračovat (opakovaným spuštěním kernelu), aniž by výsledky byly vyzvednuty. Jeden výpočetní krok simulace se tak může ještě zrychlit. Výsledky se mohou vyzvedávat např. po definovaném počtu kroků (spuštění kernelu), či na základě definované výstupní podmínky.

Vlastní nasazení NL2OCL rozšíření spočívá v instalaci NL2OCL komponent do adresářové struktury NetLoga. Java část NL2OCL.jar se nakopíruje na místo určené pro NetLogo rozšíření, konkrétně do adresáře <NetLogo instalační adresář>/app/NL2OCL/. Druhou část, dynamickou knihovnu vytvořenou v C++, NL2OCL.dll je třeba umístit přímo v adresáři <NetLogo instalační adresář> /app/.

5.3.2 Uchování sdílené informace o poloze agentů

Pro potřeby paralelního zpracování pomocí OpenCL bylo třeba nalézt vhodnou datovou strukturu, která by vyhovovala následujícím požadavkům:

- Datová struktura umožní paralelně běžícím vláknům, které reprezentující výpočty pro jednotlivé agenty, přístup k informací o poloze ostatních agentů.
- Přístup k datům v této datové struktuře musí být rychlý, nebude třeba používat žádné komplexní vyhledávání.
- Datová struktura musí umožňovat realizaci explicitního managementu paměti, tzn. strukturu bude možné naplnit daty z modelu, přenést ji do OpenCL host aplikace a odtud do OpenCL zařízení jako argonem výpočetního kernelu, kde bude zpracovávána a poté předána zpět do host aplikace a modelu.
- Datová struktura umožní konkurenční přístup, a to jak pro čtení, tak pro zápis. Bude ji používat mnoho současně běžících výpočetních vláken (reprezentujících výpočty pro jednotlivé agenty).
- Synchronizační mechanizmus nesmí být blokující (tj. založen na nějakém principu zamykání), struktura musí splňovat vlastnost „ohraničeného nečekání“ (wait-free bounded) – viz kap. 3.2.2.

Datová struktura PA (patch-agent) bude uchovávat informace o umístění všech agentů na jednotlivých buňkách prostředí modelu. Pro každou buňku prostředí modelu obsahuje PA úsek o definované kapacitě **l**. Tato kapacita určuje, jaký je maximálním možný počet agentů nacházejících se na jedné buňce modelu, o kterých datová struktura PA může uchovat informace. PA se skládá z úseků délky l umístěných za sebe. Jeden úsek odpovídá jedné buňce modelu. Hodnoty prvků pole PA obsahují buďto označení agenta (unikátní číslo agenta) nebo hodnotu -1 jako indikátor neobsazenosti pozice:

$$\begin{gathered} c_a = \text{počet agentů}, \\ 0 \leq i \leq c_a \cdot l; \qquad PA_i = \begin{cases} -1, & \text{neobsazená pozice} \\ a, & \text{a je unikátní číslo agenta na pozici i} \end{cases} \end{gathered} \tag{21}$$

Při označení souřadnic jedné buňky prostředí modelu:

$$\begin{gathered} P[x_p, y_p]; \quad x_p, x_p \in \mathbb{Z} \\ x_{p\_min} \leq x_p \leq x_{p\_max}\,, \;\; y_{p\_min} \leq y_p \leq y_{p\_max} \end{gathered} \tag{22}$$

a pokud je prostředí modelu ohraničeno minimálními a maximálními souřadnicemi buněk a jeho rozměry šířka (w) a výška (h) jsou:

$$\begin{aligned} w &= x_{p\_max} - x_{p\_min} + 1 \\ h &= y_{p\_max} - y_{p\_min} + 1 \end{aligned} \tag{23}$$

potom je celkový počet buněk c_p. Rozměry modelu a kapacita jedné buňky uchovat informaci o maximálně l agentech určují celkové rozměry pole PA a možné hodnoty indexů prvků tohoto pole:

$$\begin{gathered} c_p = w \cdot h \\ l > 0; \; l \in N \\ i \in \langle 0, c_p \cdot l - 1 \rangle; \; i \in N \end{gathered} \tag{24}$$

Každé buňce prostředí modelu odpovídá v poli obsazenosti PA jeden ohraničený úsek. Každý úsek začíná indexem, jenž je funkcí souřadnic buňky x_p, y_p, hodnot minimálních souřadnic prostředí modelu x_{p_min}, y_{p_min}, rozměrů modelu w, h a kapacity jednoho úseku l:

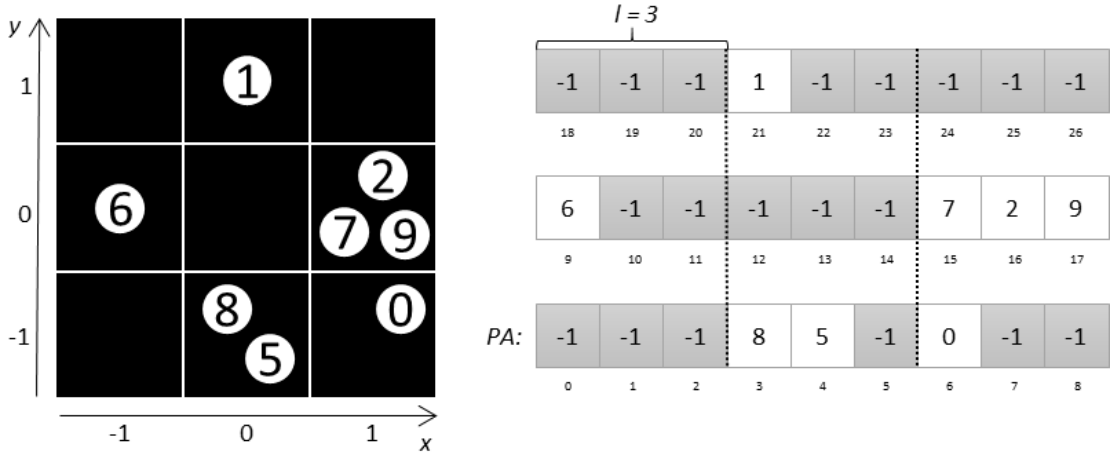
$$\begin{aligned} i_{začátek} &= f(x_p, y_p, x_{p\_min}, y_{p\_min}, w, h, l) \\ i_{začátek} &= \left[(y_p - y_{p\_min}) \cdot w + x_p - x_{p\_min} \right] \cdot l \\ i_{konec} &= i_{začátek} + l \end{aligned} \tag{25}$$

Při práci s polem obsazenosti buněk PA je třeba z indexu i jednoho prvku pole určit, jaké buňce tento prvek náleží. Souřadnice x_p, y_p jsou funkcí indexu i, hodnot minimálních souřadnic prostředí modelu x_{p_min}, y_{p_min}, šířky prostředí modelu w a kapacity jednoho úseku l. Jde o inverzní vztah ke vztahu (25):

$$x_p = f(i, x_{p\_min}, y_{p\_min}, w, l), y_p = f(i, x_{p\_min}, y_{p\_min}, w, l)$$

$$\begin{aligned} n &= \left\lfloor \frac{i}{l} \right\rfloor; \text{ číslo úseku dané buňky} \\ y_p &= \left\lfloor \frac{n}{w} \right\rfloor + y_{p\_min} \\ x_p &= \left(n - \left\lfloor \frac{n}{w} \right\rfloor \cdot w \right) + x_{p\_min} \end{aligned} \tag{26}$$

Pro ilustraci je na obr. 27 zachycena situace modelu o rozměrech 3 x 3 buňky s deseti rozmístěnými agenty. Pole obsazenosti buněk PA má při stanovené kapacitě jedné buňky l = 3 celkem 27 prvků indexovaných od 0 do 26. Pole PA je na obrázku zobrazeno tak, aby umístění jeho prvků korespondovalo s umístěním buněk modelu zobrazených v levé části obrázku (tedy od spodu nahoru). Hodnoty pole odpovídají unikátním číslům agentů nacházejících se na jednotlivých buňkách. Neobsazené pozice jsou vyjádřeny hodnotou -1.



Obrázek 27 - Pole obsazenosti buněk PA pro model o rozměrech 3 x 3 buňky, zdroj: Autor

Implementace vlastního simulačního kroku v rámci OpenCL kernelu musí obsahovat pro realizaci pohybových vzorců chování agenta kromě fyzického přesunu po souřadnicích prostředí také kontrolu přesunu agenta mezi jednotlivými buňkami. Pokud agent buňku opouští, musí dojít k aktualizaci datové struktury PA tak, aby byla zaznamenána informace o poloze agenta na nové buňce a tato informace byla prostřednictvím sdílené datové struktury PA poskytnuta ostatním agentům (jim odpovídajícím výpočetním vláknům).

Po dokončení výpočtů týkajících se rozhodnutí agenta o jeho aktuálním elementárním pohybovém kroku je toto rozhodnutí specifikováno aktuálním natočením agenta H a délkou jeho elementárního kroku s. Realizace vlastního pohybu se bude skládat z následujících programových kroků:

- výpočet složek pohybu dx a dy z daného úhlu H a délky kroku s,
- vlastní provedení pohybu – aktualizace souřadnic x = x + dx, y = y + dy,
- kontrola opuštění buňky (hranicím buňky odpovídají přesné souřadnice)
- pokud agent buňku opustil, potom:
    - aktualizace informace o současné buňce (pole PX a PY),
    - uvolnění pozice v PA struktuře, v úseku odpovídajícímu předchozí buňce (uvolnění proběhne nastavením hodnoty -1),
    - obsazení některé volné pozice v PA v rámci úseku, který odpovídá nové buňce (-1 hodnota se přepíše na hodnotu odpovídající identifikátoru agenta).

Praktická implementace tohoto procesu je dostupná v kernelu pro model HEJNA (vis příloha B).

Pro správnou funkčnost OpenCL kernelů pracujících s datovou strukturou obsazenosti buněk PA je naprosto klíčové, aby mechanizmus přesunu agentů mezi buňkami a s tím spojená aktualizace pole PA fungovaly bezchybně. Datová struktura PA zprostředkuje agentům informace o okolních agentech. Správnost chování agenta založeného na informacích o okolních agentech je tedy závislá na správnosti obsahu pole PA. Kontrolu správnosti obsahu pole PA je dobré automatizovat, vizuální/manuální kontrola hodnot pole není už pro velmi malé rozměry prostředí modelu prakticky možná.

Pro tyto potřeby byly implementovány funkce:

- FullCheckPA, QuickCheckPA – kontroly celého pole PA pro všechny buňky a agenty.
- FullCheckPAPatch, QuickCheckPAPatch – kontrola pro jednu konkrétní buňku a agenty na této buňce přítomné.
- FullCheckPAAgent, QuickCheckPAgent – kontrola pro jednoho konkrétního agenta.

Funkce FullCheckPA automaticky zkontroluje, zda obsah pole PA přesně odpovídá skutečnému umístění agentů na buňkách prostředí modelu. Prvním parametrem funkce je číslo paměťového bufferu pole PA, které bylo bufferu přiřazeno při jeho vytvoření funkcí CreatePABuffer. Funkce provádí plnou kontrolu, při které je pro každou buňku modelu porovnán seznam agentů, kteří se aktuálně na dané buňce nacházejí s hodnotami čísel agentů zapsanými v úseku pole PA, který dané buňce odpovídá. Při tomto, plném, způsobu kontroly, nelze jednoduše sekvenčně porovnat seznam agentů nacházejících se na buňce s obsahem odpovídajícího úseku pole PA. Pořadí agentů v daném úseku může být zcela odlišné a mohou se zde nacházet mezery (neobsazené pozice) obsahující hodnoty -1.

Je třeba porovnat dvě množiny čísel označujících agenty a určit, zda prvky těchto dvou množin jsou shodné. První množina A_m obsahuje jako prvky číselné identifikátory Id všech agentů nacházejících se na buňce o souřadnicích x_p, y_p. Druhá množina A_{PA} je množina prvků pole PA nacházejících se v úseku ohraničeném indexy $i_{začátek}$ a i_{konec}, které jsou pro souřadnice x_p, y_p vypočítány podle vztahu (25). Druhý parametr funkce FullCheckPA určuje směr porovnání, hodnota parametru 1 pro směr $A_m \rightarrow A_{PA}$ a hodnota parametru 2 pro směr $A_{PA} \rightarrow A_m$. Výsledkem porovnání je množina chyb E_1 – čísla agentů přítomných na dané buňce v modelu, ale chybějících v odpovídajícím úseku pole PA, E_2 – čísla agentů nacházejících se v daném úseku pole PA, ale nenacházejících se na odpovídající buňce modelu.

$$\begin{gathered} A_m = \{Id_a | \text{Agent a leží na buňce } [x_p, y_p]\} \\ A_{PA} = \{PA_i | i \in \langle i_{začátek}, i_{konec} \rangle \wedge PA_i \neq -1\} \end{gathered} \tag{27}$$

$$\begin{gathered} E_1 = A_m \setminus A_{PA} \\ E_2 = A_{PA} \setminus A_m \end{gathered}$$

Funkce QuickCheckPA neporovnává jednotlivé prvky množin A_m a A_{PA}, ale pouze kontrolní součty číselných identifikátorů agentů obsažených v obou množinách. Výsledkem porovnání je hodnota E = 0, pokud jsou součty shodné či -1, pokud jsou součty odlišné.

$$S_{AM} = \sum_{a \in A_m} a\,;\; S_{APA} = \sum_{a \in A_{PA}} a \tag{28}$$

$$E(A_m, A_{PA}) = \begin{cases} 0, & S_{AM} = S_{APA} \\ -1, & S_{AM} \neq S_{APA} \end{cases}$$

Nevýhodou tohoto způsobu kontroly je, že pouze informuje o výsledku porovnání a neposkytuje žádné další detaily. Pro rychlou kontrolu správnosti pole PA je tento způsob dostačující a je snadné po negativní odpovědi na rychlou kontrolu potřebné detaily získat provedením plné kontroly pomocí funkce FullCheckPA.

5.3.3 Práce s náhodnými čísly v OpenCL

V OpenCL verzi 2.0 není k dispozici nativně žádný generátor pseudonáhodných čísel. Výpočty v modelech se ale bez náhodných čísel neobejdou. Veškerá rozhodování založená na pravděpodobnostních modelech potřebují ke své realizaci zdroj náhodných čísel.

Obecně lze použít dva přístupy k práci s náhodnými čísly v OpenCL kernelu:

- Problém obejít – náhodná čísla v OpenCL vůbec negenerovat. Řada náhodných čísle použitelná pro běh kernelu může být připravená předem modelem a kernelu se poskytne jako argument.
- Implementovat přímo v kernelu nějaký generátor pseudonáhodných čísel.

Generátor náhodných čísel (MWC64X 2015) splňuje požadavky, které jsou od generátoru pro OpenCL vyžadovány:

- Nízké nároky – na vygenerování jednoho náhodného čísla – jsou třeba je dvě 32bitová slova.
- Rychlost – jedno vygenerování náhodného čísla obnáší vykonání pěti nebo šesti instrukcí OpenCL zařízení (podle konkrétní technologie).
- Vysoká kvalita výstupu – MWC64X generátor projde seriózními testy generátorů pseudonáhodných čísel.
- Přenositelnost – instrukce použité v MWC64X jsou k přímé dispozici na všech grafických kartách vysokého výkonu jak od AMD, tak od NVIDIE.

Vlastní implementace tohoto generátoru je dostupná v OpenCL kernelu pro model HEJNA (viz příloha B).

5.4 Funkce programového rozšíření NL2OCL

V této kapitole jsou představeny vybrané funkce poskytované programovým rozšířením NL2OCL. Výčet není úplný, uvedeny jsou pouze funkce, které jsou klíčové pro použití NL2OCL. Z uvedeného seznamu by měl čtenář získat jasnou představu o možnostech programového rozšíření NL2OCL a o způsobu jeho použití v systému NetLogo.

Funkce pro NetLogo vytvořené pomocí programového rozhraní org.nlogo.api mohou být dvojího druhu, buďto jsou kategorie reporters, to jsou funkce, které vracejí některý z NetLogem podporovaných datových typů nebo jde o kategorii commands, funkce, které nemají žádnou návratovou hodnotu. Všechny funkce NL2OCL je možné rozdělit do následujících skupin podle funkční oblasti:

- funkce pro konfiguraci prostředí běhu OpenCL aplikace,
- funkce pro manipulaci s paměťovými objekty,
- funkce pro práci s OpenCL kernely.

Tyto skupiny funkcí budou představeny v následujících podkapitolách. Pro objasnění důvodu vzniku jednotlivých funkcí je pro každou z nich uvedeno, jakého konkrétního funkčního požadavku se daná funkce týká, a to formou odkazu na jednotlivé funkční požadavky definované v kap. 5.2. Ke každé funkci

je uvedeno: jaký je její účel, zda se jedná o reporter či command, jaké má funkce vstupní parametry a jaké hodnoty vrací, jaký je proces zpracování funkce, nakonec je ukázán způsob použití.

5.4.1 Funkce pro konfiguraci prostředí běhu OpenCL aplikace

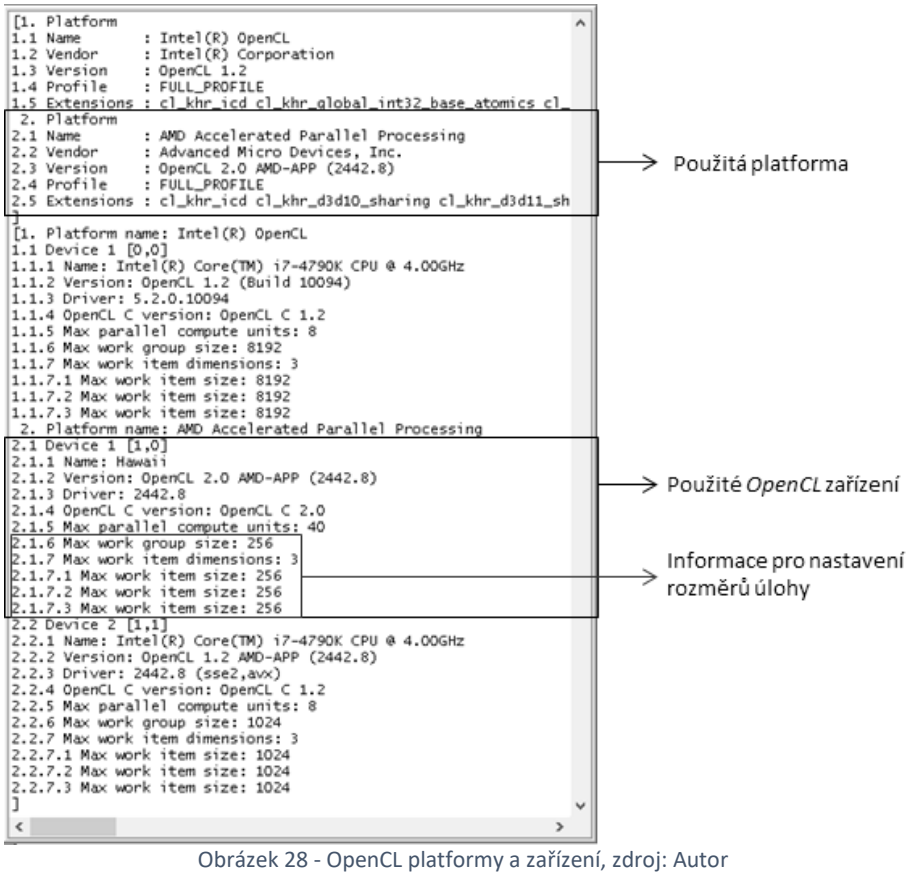
Tyto funkce umožňují nakonfigurovat prostředí běhu OpenCL aplikace přímo z modelu v systému NetLogo. Poskytují informace o dostupných prostředcích a umožňují správu zdrojů potřebných pro běh OpenCL aplikace a monitoring stavu provedení interních OpenCL funkcí.

Název:	**Typ:**	**Parametry:**	**Návratová hodnota:**
GetPlatforms GetDevices	reporter	-	pole typu String
Účel:	Funkce poskytují informace o OpenCL platformách a zařízeních, které jdou dostupné v rámci výpočetního systému, na kterém běží model v systému NetLogo. Tyto informace jsou důležité pro další nastavení prostření a paralelního výpočtu úlohy.		
Funkční požadavky:	Konfigurace prostředí běhu aplikace, **I.2**		

Použití:

```
output-print NL2OCL:GetPLatforms
output-print NL2OCL:GetDevices
```

Na obr. 28 je zobrazen příklad výstupu pro systém, na kterém běžely všechny modely představené v této práci. Na obrázku jsou vyznačeny: později vybraná platforma, vybrané OpenCL zařízení a oblast s informacemi důležitými pro správné nastavení rozměrů úlohy. Správné nastavení parametrů OpenCL kernelů: NDRange, global work group size a local workgroup size je provedeno na základě těchto parametrů.



Obrázek 28 - OpenCL platformy a zařízení, zdroj: Autor

Název:	Typ:	Parametry:	Návratová hodnota:
PrepareContextAndQueue	command	Platforma [číslo] Device [číslo]	-
Účel:	Funkce připraví programový kontext OpenCL aplikace a frontu příkazů, dva základní prvky nezbytné pro běh OpenCL aplikace. Jako vstupní parametry přijímá číslo platformy a číslo OpenCL zařízení, tato čísla jsou dostupná z výpisu platforem a zařízení pomocí GetPlatform a GetDevice.		
Funkční požadavky:	Konfigurace a správa zdrojů potřebných pro běh OpenCL aplikace, **I.1**, **I.6**		
Použití: `NL2OCL:PrepareContextAndQueue 1 0` Toto volání připraví kontext a frontu úkolů pro platformu a zařízení vyznačené u předchozího příkladu použití funkcí GetPlatform a GetDevice. Platforma: „AMD Accelerated Parallel Processing“, zařízení „Hawai“ (produktové označení čipu grafických karet rodiny R9-290 od AMD). Poznámka: vstupní parametry jsou pořadové indexy položek ze seznamů platforem a zařízení, indexy se číslují od 0.			

Název funkce:	Typ:	Parametry:	Návratová hodnota:
ClearAll	command	-	-
Účel:	Funkce uvolňuje všechny alokované prostředky a zdroje. Např. před dalším během simulace s jinou sadou parametrů a nastavení nebo při volbě jiného OpenCL zařízení.		
Funkční požadavky:	Konfigurace a správa zdrojů potřebných pro běh OpenCL aplikace, **I.1**, **I.6**		
Použití: `NL2OCL:ClearAll`			

Název funkce:	Typ:	Parametry:	Návratová hodnota:
GetLastErrorCode GetErrorMessage	Reporter	- Chybový kód OpenCL	Chybový kód OpenCL
Účel:	Obě tyto funkce zajišťují přístup k interním informacím o běhu OpenCL programu. Poskytují poslední návratové hodnoty chybových hlášení, varování a informacích z interních funkcí platformy OpenCL.		
Splněné funkční požadavky:	Zprostředkování informací o návratových hodnotách a chybových hlášení OpenCL do prostředí modelu v NetLogo. **I.7**		
Použití:			

```
let ret NL2OCL:GetLastErrorCode
if not (0 = ret) [
  show NL2OCL:GetErrorMessage ret
]
```

Pokud návratová hodnota posledního interního volání nějaké OpenCL funkce nebyla 0 (konstanta CL_OK), zobrazí textovou zprávu odpovídající tomuto chybovému kódu.

5.4.2 Funkce pro manipulaci s paměťovými objekty

Tyto funkce slouží k plnění úkolů explicitního managementu paměti v OpenCL aplikacích, který je detailně popsán v kapitole 5.3.1.

Při implementaci prototypu NL2OCL nejprve vznikly funkce pro správu paměťových objektů, které, jednotlivé kroky managementu paměti provádějí v detailních, separátních krocích. Management paměti je tak sice pod detailní kontrolou, ale na druhou stranu použití může být technicky komplikovanější a zavádí do kódu vlastního modelu části kódu, které mohou odvádět pozornost od vlastního modelu. V rámci druhé vývojové fáze NL2OCL došlo k podstatnému zapouzdření celého procesu manipulace s paměťovými objekty. Zodpovědnost za provádění jednotlivých kroků managementu paměťových objektů a datových struktur přešla pod správu NL2OCL. Byly implementovány následující funkce, které management datových objektů mezi jednotlivými částmi heterogenního výpočetního systému NetLogo – NL2OCL – OpenCL podstatně usnadňují.

Funkce mají jména tohoto charakteru: Register**Int**BufferHolderFrom**AS**. Tato jména v sobě obsahují informace o tom, jakých datový typů se týkají (v tomto případě Integer) a odkud čerpají data pro zřizované paměťové buffery (v tomto případě AS – množina agentů, Agent Set). V následující tabulce je přehled základních registračních funkcí:

Tabulka 7 - Konstrukce jmen registračních funkcí paměťových bufferů programového rozšíření NL2OCL, zdroj: Autor

	Datový typ			Množina dat	Funkce vytváří a synchronizuje:
Register	**Int**	BufferHolder	From	**AS**	Paměťový buffer, jehož prvky jsou datového typu Integer vytvořený z množiny agentů (agent set).
	Float			**AS**	Float – z množiny agentů (agent set).
	Int			**PS**	Integer – z množiny buněk (patch set).
	Float			**PS**	Float – z množiny buněk (patch set).
	Int			**Val**	Integer – prvky bufferu jdou vypleněny předepsanou hodnotou.

	Float			**Val**	Float – prvky bufferu jdou vypleněny předepsanou hodnotou.
	Int			**List**	Int – prvky bufferu jdou vyplněny hodnotami ze zadaného NetLogo listu.
	Float			**List**	Float – prvky bufferu jdou vyplněny hodnotami ze zadaného NetLogo listu.
	Breed		-	-	Multi-registrační funkce pro více atributů jednoto typu agentů (breed)
	AgentSet		-	-	Multi-registrační funkce pro více atributů specifikované množiny agentů (breed)

Zde je popis a ukázka nejdůležitějších registračních funkcí:

Název:	**Typ:**	**Návratová hodnota:**
RegisterIntBufferHolderFromAS RegisterFloatBufferHolderFromAS RegisterIntBufferHolderFromPS	Reporter	Číselný identifikátor označující objekt správce paměťového bufferu.
Účel:	Tyto funkce provedou registraci k automatizaci paměťového managementu ve směru NetLogo model → OpenCL aplikace →OpenCL zařízení (kernel argument). Pro zvolenou množinu agentů nebo buněk a zvolený atribut (atribut může být jak vestavěný, tak přidaný uživatelsky) se vytvoří paměťový buffer odpovídajícího datového typ, do kterého se zkopírují hodnoty daného atributu. Buffer je potom připravený k automatickému použití jakožto argumentu zvoleného kernelu.	
Splněné funkční požadavky:	Management paměti základní požadavky – **II.1-4** Pokročilé požadavky – **II.7** a **II.9**	
Parametry:	• AgentSet nebo PatchSet – vybraná množina agentů nebo buněk, pro které se vytvoří „registrace“. • String – jméno atributu agenta nebo buňky (např. „xcor“), ze kterého se budou získávat hodnoty pro paměťový buffer, může jít i o název uživatelsky zavedeného atributu. • Integer – číselné označení funkce kernelu, pro kterou bude vytvořený paměťový buffer použit jako argument. • Integer – index argumentu funkce kernelu, ve kterém bude vytvořený paměťový buffer použit při spuštění kernelu.	

	• Integer – požadavek na zpětnou synchronizaci (1→ synchronizovat, 0 → nesynchronizovat).

Použití:
`let heading_buf NL2OCL:RegisterIntBufferHolderFromAS pedestrians "heading" 0 0 0`
`let xcor_buf NL2OCL:RegisterFloatBufferHolderFromAS pedestrians "xcor" 0 1 1`
`let ycor_buf NL2OCL:RegisterFloatBufferHolderFromAS pedestrians "ycor" 0 2 1`
`let myattr_buf NL2OCL:RegisterIntBufferHolderFromPS patches with [color = red] "my_attribute" 0 3 0`
První volání funkce RegisterIntBufferHolderFromAS připraví paměťový buffer pro atribut heading všech agentů typu pedestrian, tento paměťový buffer bude použit pro první funkci kernelu (číslo 0) jako jeho první (0-tý) argument. Na vytvořený paměťový buffer nejsou synchronizační požadavky (poslední parametr funkce je 0). Druhé a třetí volání je funkce RegisterFloatBufferHolderFromAS. Pro stejnou množinu agentů jako v prvním případě se připraví paměťové buffery, tentokrát pro datový typ float. Budou reprezentovat hodnoty atributů xcor a ycor – Tyto paměťové buffery budou použity také pro první funkci kernelu, a to jako jeho druhý (označení 1) a třetí (označení 2) argument. Poslední parametr je v obou případech 1, což znamená, že bude vyžadována synchronizace – vyzvednutí hodnot z bufferů a jejich zpětné nastavení do atribut xcor a ycor agentů typu pedestrian.	

Předchozí registrační funkce poskytují zapouzdření první části explicitního managementu paměti, tj. cesty dat od atributů agentů a buněk přes OpenCL aplikaci k OpenCL zařízení (prostřednictvím atributu kernelu, který je typu paměťový buffer). Tyto funkce tak vedou k odstranění nadbytečně detailního managementu paměťových objektů a k celkovému zjednodušení částí kódu NetLogo modelu, které souvisejí s použitím programového rozšíření NL2OCL.

Následující funkce jdou v procesu automatizace kroků spojených s výměnou dat mezi NetLogem a OpenCL ještě o krok dále a dovolí uživateli nastavit vše potřebné pro management paměťových bufferů jediným příkazem.

Název:	**Typ:**	**Návratová hodnota:**
RegisterBreedBuffersHolder RegisterAgentSetBuffersHolder	Reporter	Číselný identifikátor označující objekt správce množiny paměťových bufferů.
Účel:	Kombinovaná multi-registrace k automatizované, oboustranné datové výměně mezi NetLogem, NL2OCL a OpenCL. Vytvoří se paměťové buffery pro všechny agenty zvoleného typu (RegisterBreedBuffersHolder) nebo pro vybranou skupinu agentů (RegisterAgentSetBuffersHolder). Je možné najednou zaregistrovat více atributů a požadavků na jejich synchronizaci.	
Splněné funkční požadavky:	Management paměti – pokročilé požadavky **II.7-9, II.11**	
Parametry:	• String – jméno typu agentů (breed).	

	• Integer – číselné označení funkce kernelu, pro kterou budou vytvořené paměťový buffery použity jako argumenty. • StringList – jména registrovaných atributů • StringList – seznam požadovaných datových typů pro jednotlivé paměťové buffery (hodnoty "F" – pro datový typ float a "I" pro integer) • IntegerList – seznam indexů argumentů funkce kernelu, ve kterých budou vytvořené paměťové buffery použity při spuštění kernelu. • IntegerList – seznam synchronizačních požadavků na jednotlivé zaregistrované atributy (1→ synchronizovat, 0 → nesynchronizovat).
Použití: `set bh NL2OCL:RegisterBreedBuffersHolder "pedestrians" 0 ["xcor" "heading" "ycor" "my_attribute"] ["I" "F" "I" "F"] [0 1 2 3] [1 0 1 0]` Pro typ agentů „pedestrians" se zaregistrují pro první funkci kernelu (označení 0) celkem čtyři paměťové buffery pro vestavěné atributy xcor, heading, ycor a uživatelsky přidaný atribut my_attribute. Čtvrtý parametr funkce ["I" "F" "I" "F"] říká, že buffery budou obsahovat data typu integer (první buffer pro xcor a třetí buffer pro ycor) a dva buffery budou obsahovat data typu float (druhý buffer pro atribut heading a čtvrtý buffer pro uživatelský atribut my_heading). Paměťové buffery budou použity jako argumenty kernelu s indexy 0 až 3 (pořadí argumentů kernelu [0 1 2 3] může být nicméně definováno 5 tým argumentem funkce libovolně). Poslední parametr funkce [1 0 1 0] udává, že první a třetí paměťové buffery budou vyžadovat zpětnou synchronizaci po dokončení výpočtu kernelu, zatímco druhý a čtvrtý paměťový buffer budou sloužit jen jako argumenty kernelu, ale nebudou se synchronizovat zpět do NetLoga.	

Další skupina funkcí usnadňuje druhou polovinu explicitního managementu paměti, tj. zpětnou cestu dat z OpenCL zařízení, přes OpenCL aplikaci do modelu v systému NetLogo (v OpenCL). Slouží primárně k vyzvedávání hotových výpočtů. A to zejména v případech, kdy výsledné hodnoty není třeba ihned nastavit jako atributy agentů či buněk či pokud je automatizace tohoto kroku nežádoucí, např. pokud je plánováno, že se s daty na úrovni polí hodnot bude v NetLogo modelu ještě nějak přímo manipulovat před použitím pro atributy agentů a buňky.

Název:	**Typ:**	**Návratová hodnota:**
GetIntBufferFromBH GetFloatBufferFromBH GetIntPatchBufferFromBH	Reporter	NetLogo list obsahující hodnoty z paměťového bufferu, o který se stará správce daného čísla
Účel:	Funkce provádějí druhou část explicitního paměťového managementu vyžadovaného v OpenCL aplikacích. To znamená, že převezmou paměťové objekty z fyzického paměti OpenCL zařízení (provede NL2OCL.dll) a zkopírují je do OpenCL host aplikace, kde se připraví k prezentaci jako pole hodnot	

	odpovídajícího typu (provede NL2OCL.jar). Od je pole navráceno jako NetLogo list do prostředí běhu modelu.
Splněné funkční požadavky:	Management paměti **II.5, II.9**
Parametry:	Integer – číslo paměťového správce bufferu přidělené při registraci daného buffer holderu.

Použití:
`set i_data NL2OCL:GetIntBufferFromBH bh_01`
`set f_data NL2OCL:GetFloatBufferFromBH bh_02`
Vyzvedne dvě pole hodnot, první typu Integer, druhé typu Float, z paměťových bufferů spravovaných prostřednictvím správců s číselným označením bh_01 a bh_02, a nastaví hodnoty jako NetLogo listy i_data a f_data.	

Pokud nás v předchozí sadě funkcí zajímaly paměťové buffery jako pole hodnot, s nimiž budeme ještě nějak manipulovat, následující skupina funkcí reprezentuje úplnou automatizaci managementu paměti. Funkce využívají plně možností správců paměťových bufferů, které byli v rámci tvorby NL2OCL implementovány. Funkce automaticky synchronizují obsah paměťových bufferů z OpenCL kernelů do hodnou atributů agentů a buněk na základě konfigurace provedené registračních funkcí představených výše.

Název:	**Typ:**	**Návratová hodnota:**
SynchonizeIntBHToAttributes SynchonizeIntBHToAttributes SynchonizeAllBHToAttributes	Command	-
Účel:	Funkce automaticky synchronizují atributy agentů nebo buněk s registrovanými paměťovými buffery. Dojde k vyzvednutí hodnot z kernelu OpenCL, zpracování OpenCL aplikací, atributy agentů/buněk se provede přímo v NL2OCL v modelu NetLoga již není potřeba provádět žádné další kroky. Do jakých atributů jsou hodnoty synchronizovány, to je specifikováno při registraci příslušného správce paměťového bufferu.	
Splněné funkční požadavky:	Paměťový management – II**.4, II.5** Pokročilé požadavky na automatizaci – II**.9**	
Parametry:	Integer – Číslo správce paměťového bufferu	
Použití: `NL2OCL:SynchonizeIntBHToAttributes bh_01` Provede se synchronizace hodnot paměťového bufferu do atributu specifikovaného při registraci (pomocí funkce RegisterIntBufferHolderFromAS) a to pro agenty, kteří byli také při registraci specifikováni (množinou agentů – agent setem).		

K těmto funkcím, které provádějí synchronizaci specifických paměťových bufferů, existuje ještě jedna další funkce, která jejich funkcionalitu sdružuje a provádí synchronizaci pro všechny registrované paměťové buffery najednou.

Název:	**Typ:**	**Návratová hodnota:**
SynchonizeAllBHToAttributes	Command	-
Účel:	Funkce postupně provede synchronizaci všech paměťových bufferů, pro které je synchronizace požadována. Typicky proběhne volání této funkce jako jediné volání synchronizace po úspěšném vykonání provedení funkce kernelu.	
Splněné funkční požadavky:	Pokročilé požadavky na automatizaci synchronizace – II**.9**	
Parametry:	-	
Použití: `NL2OCL:SynchonizeAllBHToAttributes` Funkce provede veškerou synchronizační práci. Nemá žádné parametry ani nevrací žádnou návratovou hodnotu. Pokud dojde k nějaké chybové události, uživatel se do dozví prostřednictvím výjimky NL2OCLException, ve které budou uvedeny všechny potřebné detaily.		

5.4.3 Funkce pro práci s OpenCL Kernely

Příprava a spouštění OpenCL kernelů je základním prostředkem programového rozšíření NL2CL, jak jsou realizovány paralelizované výpočty. Pro spuštění kernelů je třeba nastavit jejich argumenty – těmi jsou jednak paměťové buffery připravené funkcemi popsanými v předchozí kapitole a další argumenty, které nemají povahu polí (většinou jde o konstanty používané při výpočtu a parametry modelu). Kroky životního cyklu kernelů jsou detailně popsány v kapitole 5.7.2. Funkce pro zajištění těchto kroků jsou následující:

Název:	**Typ:**	**Návratová hodnota:**
PrepareKernelFunctions	Command	-
Účel:	Tato funkce projde specifikovaný vstupní soubor se zdrojovým kódem kernelu a připraví funkce kernelu ke spuštění. Nejprve jsou identifikovány a očíslovány funkce kernelu (jsou číslovány od 0, tato čísla jsou použitá ve všech registračních funkcí pro paměťové buffery). Zdrojový kód kernelu je zkompilován pomocí OpenCL kompilátoru a sestaven pro spuštění na konkrétním OpenCL specifikované při vytvoření kontextu OpenCL aplikace a fronty úkolů.	
Splněné funkční požadavky:	Správa kernelů základní funkčnost – **III.2** Pokročilé požadavky – **III.7-9**	

Parametry:	String – název souboru se zdrojovým kódem funkcí kernelu (napsaným v C-OpenCL). Funkce očekává, že daný soubor bude ve stejném adresáři odkud je spuštěný NetLogo model.

Použití:

```
NL2OCL:ClearAll
NL2OCL:PrepareContextAndQueue 1 0
NL2OCL:PrepareKernelFunctions "pedestrian_evacuation.cl"
```

Příprava kernelu může proběhnout až v době, kdy jsou připravené kontext OpenCL aplikace a fronta úkolů. Ty jsou totiž svázány s vybranou platformou a OpenCL zařízením. Pro toto OpenCL zařízení probíhá sestavení programu po jeho úspěšné kompilaci – výsledný binární kód kernelu se může pro různá OpenCL zařízení lišit. Uvedená sekvence tří příkazů připraví OpenCL k běhu a kernel ke spuštění.

Název:	**Typ:**	**Návratová hodnota:**
RunKernel	Command	Návratová hodnota OpenCL (0 pro úspěšně provedený kernel, záporná hodnota, pokud dojde k nějaké chybě)
Účel:	Provedení připraveného kernelu (resp. jedné funkce kernelu)	
Splněné funkční požadavky:		
Parametry:	Integer – identifikační číslo funkce kernelu (získané při kompilaci a sestavení kernelu), Integer – dimenze řešené úlohy, číslo 1 až 3 (závisí na tom, jak je navržen kernel a na možnostech OpenCL zařízení, pro které je zřízen OpenCL aplikační kontext a fronta úloh), Integer pole – Globální velikost (pole má tolik prvků jako je dimenze úlohy), určuje celkový počet vláken, která budou spuštěna pro jednotlivou dimenzi úlohy, Integer pole – lokální velikost (pole má tolik prvků jako je dimenze úlohy), určuje velikost pracovní skupiny vláken pro jednotlivé dimenze.	

Použití:

Funkce zařadí požadavek na zpracování kernelu do asynchronní fronty úloh OpenCL aplikačního kontextu, ve kterém je kernel vytvořen. Samo volání funkce je synchronní (NL2OCL zapouzdřuje vyčkání na dokončení kernelu) a vrací OpenCL návratovou hodnotu. Pokud je tato hodnota rovna 0, kernel byl zpracován správně, pokud je hodnota záporná, znamená to, že došlo k nějaké chybě (např. -52 = CL_INVALID_KERNEL_ARGS, špatné nastavení argumentů kernelu). Pomocí funkce NL2OCL:GetErrorMessage <kód chyby> je možné získat textovou interpretaci chybového hlášení.

6 Paralelizace vybraných modelů

Navržený metodický postup pro paralelizaci multi-agentových modelů a jeho technická realizace byly ověřovány na různých modelech, z nichž jsou v této práci z prostorových důvodů představeny tři v následujících kapitolách 6.1, 6.2 a 6.3.

6.1 Model HEJNA – paralelizace Reynoldsova modelu hejna

Jedná se o klasický model hejna (Reynolds 1987), který existuje jako knihovní model NetLoga (Wilensky 1998). Model byl vybrán k demonstraci použití metodického postupu paralelizace a k ukázce použití NL2OCL programového rozšíření především proto, že chování modelu je předem známé a lze dobře ověřit, že paralelizovaná veze modelu pracuje správně. Tento model je příkladem multi-agentového modelu se spojitými souřadnicemi, ve kterém je chování agentů složeno z rozdílných rozhodovacích vzorcích (align, cohere, separate). Dalším důvodem výběru tohoto modelu byla skutečnost, že na tomto modelu bylo možné ukázat sadu základních výpočetních prostředků, které je třeba k paralelizaci mnohem složitějších modelů. Vytvořený paralelizovaný model, zejména jeho část implementovaná jako OpenCL kernel, obsahuje fragmenty ihned použitelné v dalších, mnohem komplikovanějších, modelech.

6.1.1 Návrh

Specificky má model hejna tyto vlastnosti, které se odrazili ve způsobu implementaci paralelně pracující varianty tohoto modelu:

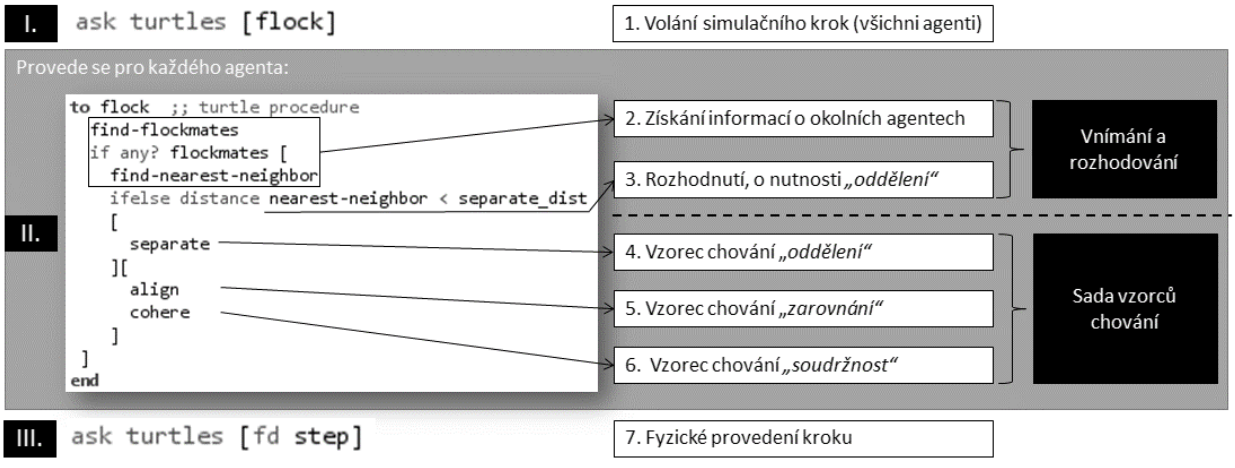
- Chování každého agenta je kompozicí tří oddělených rozhodovacích vzorců. Na modelu tak zle demonstrovat, jak je možné několik nezávislých rozhodovacích vzorců chování převést do paralelní podoby. Jedná se o tyto vzorce chování:
    - Oddělení (separation) – vzorec chování vedoucí k vyhnutí se přílišnému seskupení.
    - Zarovnání (alignment) – vzorec chování pro adaptaci pohybu agenta ve směru průměrného směřování okolích agentů.
    - Soudržnost (cohesion) – vzorec chování zabezpečující seskupení více agentů pohybem směrem k průměrné pozici okolních agentů.
- Pohyb agentů v modelu je spojitý – nepohybují se po diskrétních buňkách, ale po celé ploše prostředí modelu v krocích definované délky. Agenti se mohou nacházet obecně na jakýchkoli souřadnicích. Práce s takovýmto typem modelu vyžaduje použití speciální sdílené datové struktury pro uchování informace o poloze agentů na jednotlivých buňkách. V modelu je demonstrováno použití datové struktury PA, která je detailně popsána v kapitole 5.3.2 a je ukázáno, jak s touto datovou strukturou pracovat za pomoci atomických operací z OpenCL tak, aby byla bezpečná pro

konkurenční čtení i zápis a měla vlastnost neblokující datové struktury (vlastnost popsaná detailně v kap. 3.2.2).

- Realizace pohybových vzorců chování v tomto modelu vyžaduje speciální druh výpočtů s polohami agentů a s úhly jejich natočení. Jde o typické úlohy, jejichž implementace je na tomto modelu demonstrována a může být použita ve všech dalších modelech používajících tento druh výpočtů. Jedná se o tyto úlohy:
- Převod geometrických úhlů na úhly použitelné v prostředí modelu v NetLogu. Úhly se v NetLogu měří jinak, než jak je tomu standardně v kartézské soustavě souřadnic, kde se úhly měří od osy x proti směru chodu hodinových ručiček. V NetLogu se úhly měří od směru „nahoru“ (označeno jako sever) a to po směru hodinových ručiček.
- Rozsah simulace se dá jednoduše zvětšovat, jak zvětšováním velikosti prostředí modelu, tak i zvětšováním populace agentů. Lze tak dobře demonstrovat rozdílnou výkonnost simulace při použití paralelního zpracování prostřednictvím programového rozšíření NL2OCL oproti výkonnosti dostupné jen za pomoci výpočtů realizovaných systémem NetLogo.

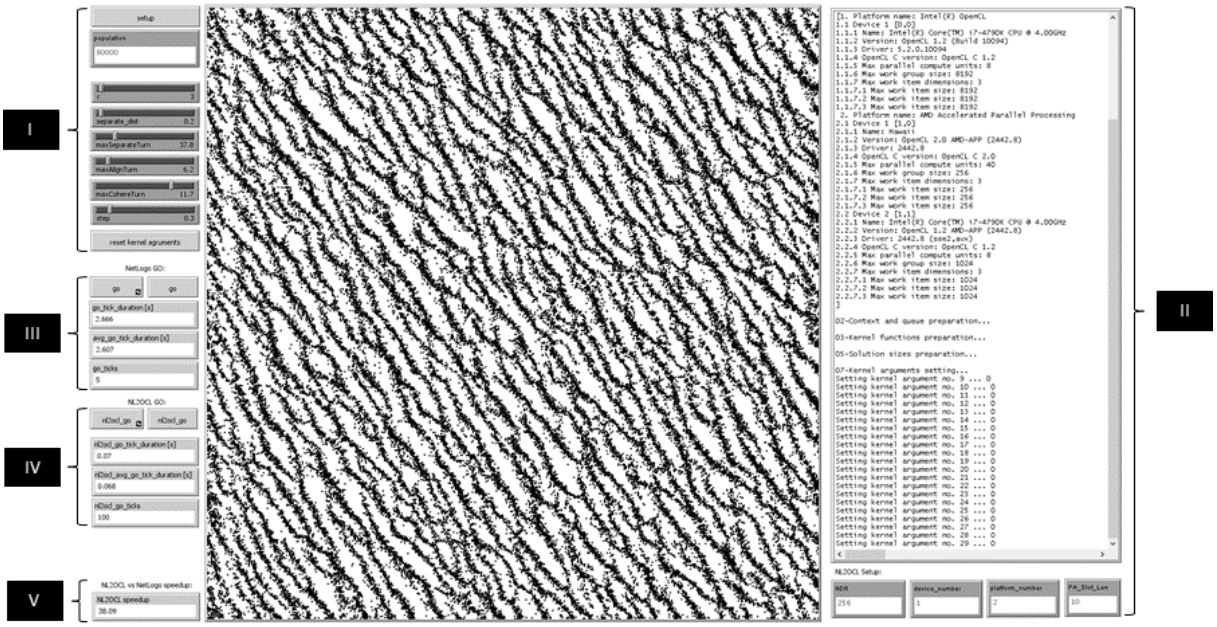
6.1.2 Implementace

Na obr. 29 je zachycena struktura hlavní výkonné části kódu modelu v NetLogu. Obrázek ukazuje, jakým způsobem bude probíhat paralelizace tohoto modelu. Jeden simulační krok volaný částí I se skládá z bloku, který se opakovaně provádí pro všechny agenty – část II, část III už jen vykoná připravený pohyb. Část II je to, co je možné zpracovat paralelně. Funkčnost této části je implementována jako OpenCL kernel – viz příloha B (výpis OpenCL kernelu pro model HEJNA).



Obrázek 29 - Model HEJNA, struktura simulačního kroku v NetLogu, body k paralelizaci, zdroj: Autor

Na obr. 30 je zobrazeno uživatelské rozhraní paralelizovaného modelu HEJNA v NetLogu. Jednotlivé části, které jsou označené římskými číslicemi, jsou obecně použitelné (spolu s NetLogo programovým kódem, který k nim náleží) i v dalších modelech využívajících programové rozšíření NL2OCL.



Obrázek 30 – Model HEJNA, uživatelské rozhraní modelu v NetLogu, zdroj: Autor

Toto je funkční význam jednotlivých částí uživatelského rozhraní modelu HEJNA na obr. 30:

I. Nastavení parametrů modelu, vizuálního dosahu agenta r, minimální vzdálenost agentů (při menší vzdálenosti dojde k oddělení), maximální úhel, pod kterým se agent může oddělit od hejna (maxSeparateTurn), maximální úhel, pod kterým se agent v rámci jednoho simulačního kroku může zarovnat do hejna (maxAlignTurn), maximální úhel pro udržování soudržnosti (maxCohereTurn). V této sekci se nachází také tlačítko „Setup“. Stisk tlačítka připraví iniciální stav simulace. Pomocí NL2OCL rozšíření připraví prostředí OpenCL aplikace pro vlastní běh (na základě konfigurace v sekci II vytvoří OpenCL aplikační kontext, frontu úloh, zkompiluje a sestaví kód kernelu, připraví paměťové buffery a nastaví argumenty kernelu).

II. Konfigurace OpenCL prostředí, výpis OpenCL platforem a zařízení, nastavení rozměrů a rozsahu úlohy, které jsou použity při spouštění kernelů (ND Range, Local/Global size).

III. Spouštění izolovaného simulačního kroku počítaného jen pomocí prostředků NetLogo, časové trvání takovýchto simulačních kroků (aktuální, průměrné) nebo spuštění kontinuálního běhu simulace.

IV. Spouštění izolovaného simulačního kroku počítaného pomocí programového rozšíření NL2OCL, časové trvání takovýchto simulačních kroků (aktuální, průměrné) nebo spuštění kontinuálního běhu simulace.

V. Údaj o zrychlení simulace při použití NL2OCL oproti simulaci provedené NetLogem.

6.1.3 Experimenty

Porovnání dosažených zrychlení při použití NL2OCL oproti provedení simulace jen pomocí prostředky NetLoga, pro různé velikosti populace agentů a pro různé vizuální poloměry agenta R (vizuální poloměr je vzdálenost, na kterou agent vnímá okolní členy hejna).

Postup experimentu – Model byl nastaven tak, aby prostředí NetLogo pracovalo co nejrychleji, tzn. kontrolní prvek umožňující měnit rychlost simulace byl nastaven na maximum. Dále bylo vypnuto automatické aktualizování grafické informace v prostředí modelu. V tomto experimentu šlo o porovnání maximálních možných výpočetních rychlostí, kdy ukazatelem byla doba trvání výpočtů potřebných pro uskutečnění jednoho simulačního kroku, vykreslování aktualizací by tyto časy zkreslovalo.

Iniciální parametry modelu – Postupně byl nastaven vizuální poloměr R na hodnoty 3, 8 a 15 a pro každý vizuální poloměr byly nastaveny počáteční populace 10 000 až 100 000 agentů. To znamená celkem 30 iniciálních stavů simulace. Pro každý iniciální stav byla simulace spuštěna ve dvou variantách. Při první variantě provádělo výpočty pouze NetLogo, ve druhé variantě byly výpočty prováděny paralelně pomocí NL2OCL. Byla měřena průměrná doba trvání jednoho simulačního kroku. Výsledky experimentu jsou zobrazeny v tab. 8 a v grafu na obr. 31.

Tabulka 8 – Přehled dosažených zrychlení simulace s modelem HEJNA pro různé populace agentů a pro různý vizuální poloměr agenta R, zdroj: Autor

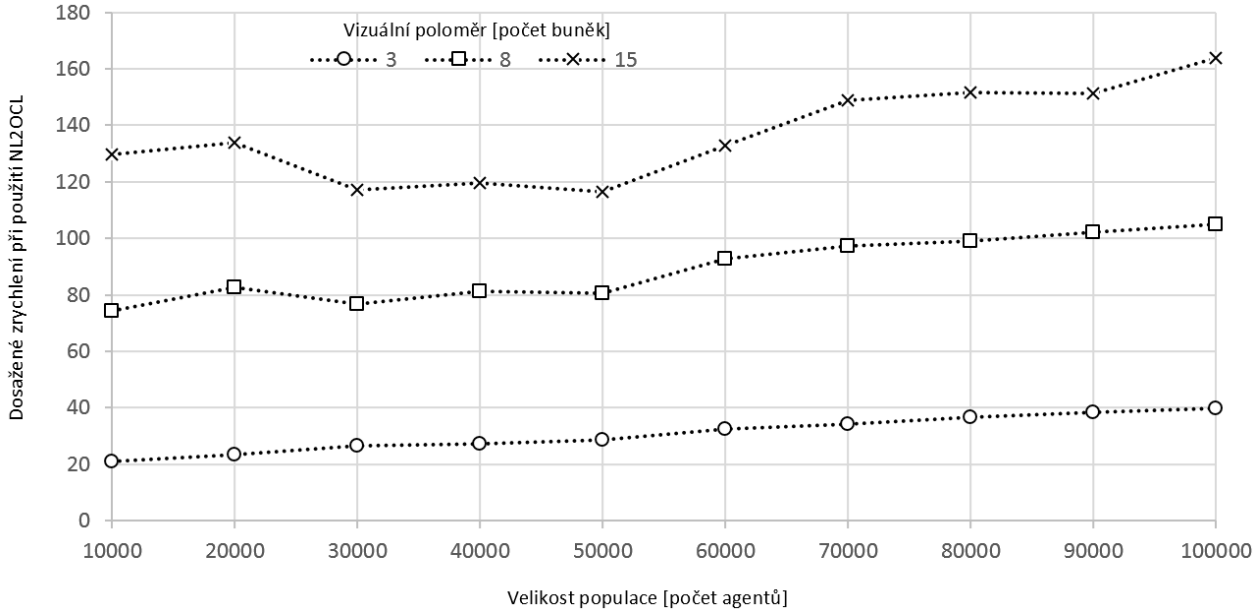
R			Počet Agentů									
			10 000	**20 000**	**30 000**	**40 000**	**50 000**	**60 000**	**70 000**	**80 000**	**90 000**	**100 000**
3	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	0,125	0,279	0,529	0,762	1,029	1,399	1,771	2,194	2,607	3,067
		NL2OCL	0,006	0,012	0,02	0,028	0,036	0,043	0,052	0,060	0,068	0,077
	Zrychlení simulačního kroku		**21**	**23**	**26**	**27**	**29**	**33**	**34**	**37**	**38**	**40**
8	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	0,521	1,407	2,223	3,743	5,387	6,682	8,653	10,803	13,067	15,662
		NL2OCL	0,007	0,017	0,029	0,046	0,067	0,072	0,089	0,109	0,128	0,149
	Zrychlení simulačního kroku		**74**	**83**	**77**	**81**	**80**	**93**	**97**	**99**	**102**	**105**
15	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	1,555	3,884	6,554	10,393	14,907	17,543	22,936	28,209	35,099	46,739
		NL2OCL	0,012	0,029	0,056	0,087	0,128	0,132	0,154	0,186	0,232	0,285
	Zrychlení simulačního kroku		**130**	**134**	**117**	**119**	**116**	**133**	**149**	**152**	**151**	**164**

Interpretace výsledků experimentu (na základě naměřených dat a jejich grafické vizualizace):

- Při použití NL2OCL roste zrychlení simulace jednak se zvětšující se populací a také při zvětšujícím se vizuálním poloměru. Oba tyto parametry ovlivňují celkový počet výpočtů, které

je třeba provést. To, že zrychlení roste při zvětšujícím se rozsahu úlohy svědčí o dobré škálovatelnosti implementovaného řešení.

- Velikost populace ovlivňuje rozsah úlohy lineárním způsobem. Pro každého dalšího agenta je třeba provést stejné množství individuálních výpočtů jako pro ostatní agenty. Tento fakt je možné z grafu vypozorovat – tři grafy pro různé vizuální poloměry mají stejné tempo růstu.
- Pro zvětšující se vizuální poloměr roste rozsah úlohy kvadraticky. Větší vizuální poloměr znamená větší čtvercové okolí agenta, ve kterém se provádějí výpočty (výpočet začíná nalezením kruhového okolí ve čtvercové oblasti).
- Pro vizuální poloměry 8 a 15 je možné v grafu na obr. 31 pozorovat pokles zrychlení pro populace 30 000 až 60 000 agentů. Tento pokles souvisí s výkonem grafické karty, který se musel dělit mezi výpočty pro NL2OCL a jiné úlohy běžícího systému (ke grafické kartě byl připojen monitor). Při opakovaném měření ke snížení výkonu nedošlo. Promítnutí této události do výsledků experimentu je ponecháno záměrně, má totiž důležitou vypovídající hodnotu. Pro konfiguraci heterogenního výpočetního systému, jehož výkon by byl plně pod kontrolou by byla zapotřebí konfigurace s dedikovanou grafickou kartou, kterou by neúkoloval žádný jiný proces.



Obrázek 31 – Dosažená zrychlení simulace v modelu HEJNA při použití NL2OCL pro 10 000 až 100 000 agentů a vizuální poloměr 3, 8 a 15, zdroj: Autor

6.2 Model EVAKUACE – paralelizace evakuačního modelu chodců

6.2.1 Návrh

Model je určen pro provádění rychlých simulací evakuace velkého počtu osob (cca 100 000) z uzavřeného prostředí s jedním východem. Model je navržen jako demonstrační model pro paralelní

zpracování modelu typu buněčného automatu. Může sloužit jako rychle pracující stavební prvek (sub-model) pro evakuační modely s komplikovanějším prostředím (tvar, překážky, více východů) a/nebo s komplikovanějšími rozhodovacími procesy agentů reprezentujících osoby.

Entity: V modelu existují tyto třídy agentů:

- Agenti – Evakuované osoby (v modelu mají označení Pedestrian)
- Buňky prostředí – které reprezentují volnou plochu, zeď/překážku nebo dveře (v modelu mají označení Cell).

Atributy (stavové veličiny): Agent reprezentující v modelu osobu má tyto atributy:

- umístění – souřadnice buňky, na které se agent nachází,
- poslední směr pohybu – identifikátor směru pohybu během posledního simulačního kroku,
- rychlost – počet vykonaných kroků za jednotku času.

Atributy prostředí jsou:

- S – statický ukazatel vzdálenosti buňky od východu,
- D – dynamický ukazatel, jak je buňka v čase agenty využívána,
- M – ukazatel stupně usilování o buňku,
- Obsazenost buňky – volá/obsazená, zeď, dveře.

Časové a prostorové rozměry: Čas simulace plyne v krocích, jeden simulační krok je doba trvání rozhodnutí všech osob o dalším kroku a provedení tohoto kroku. Běh simulace trvá od zahájení do úplného vyprázdnění prostoru.

Hexagonální síť, která se v modelu používá, je vytvořena vzájemným posunutím sudých a lichých řad buněk o vzdálenost poloviny buňky. Prostorové rozměry jsou i v hexagonální síti stejné. Hrana jedné čtvercové buňky odpovídá 40 cm reálné vzdálenosti. Kroky agentů v hexagonální síti budou také 40 cm. Např. při volbě rozměrů 300 x 300 čtvercových buněk systému NetLogo, které budou odpovídat reálné čtvercové ploše 75 x 75 m. Do takového prostředí bude možné umístit populaci až 90 000 agentů.

Principy, teorie: Základní princip modelu vychází z modelu chodců jako buněčného automatu využívajícího statické a dynamické „silové“ pole podlahy, který byl prezentován v (Burstedde at al., 2001). Dále jsou využity vylepšení modelu popsaná v (Yang at al., 2015) týkající se výpočtů hodnot statického a dynamického pole pro jednotlivé buňky. Obě varianty modelů využívají čtvercovou síť buněk.

CA modely chodců mohou využívat čtvercovou síť buněk. Výhodou tohoto uspořádání je jednodušeji uspořádané okolí buněk a s tím související jednodušší výpočty v buněčném automatu. Množinu okolních buněk, na které je možné se z dané buňky přesunout, tvoří buďto čtveřice buněk sousedících v kolmých směrech (tzv. Neumannovo okolí), nebo všech osm sousedních buňek. Osmibuňkové okolí má velkou nevýhodu v tom, že délka kroku v kolmých směrech je odlišná od délky kroku ve směrech diagonálních. Tím je do modelu vnášena nepřesnost. Tuto nepřesnost je možné řešit zavedením buněčné sítě, která je hexagonální. Ta disponuje ve všech šesti možných směrech pohybu stejnou vzdáleností. Model s hexagonální sítí buněk je ale výpočetně náročnější.

Rozhodování agentů: Kombinací výše zmíněných modelů docházíme ke kombinaci rozhodovacích faktorů. Jejich použití bude zaručovat dostatečnou přesnost modelu. Navíc bude tato kombinace při použití v hexagonální buněčné síti dostatečně výpočetně náročná tak, aby se projevily výhody paralelního zpracování výpočtů modelu pomocí NL2OCL.

Těmito rozhodovacími faktory jsou:

- **Sledování nejkratší cesty k východu –** S
    - Agenti vnímají požadovaný směr nejkratší cesty prostřednictvím hodnot S statického „silového“ pole podlahy. Hodnoty S jsou nastaveny tak, že hodnota S_C buňky C určuje, jaká je minimální vzdálenost měřená v počtu buněk mezi danou buňkou a východem. Nastavení této hodnoty lze provést při inicializaci tak, že se všem buňkám nastaví nejprve hodnota S = 0, poté se buňkám, které reprezentují východ přiřadí hodnota 1 a postupně se pro všechny buňky s nenulovou hodnotou S hledají jejich sousední buňky s hodnotou S = 0 a těm se jako S přiřadí hodnota o jednu větší – hodnota S se tedy vlnou postupně rozšíří po celém prostředí modelu. Agenti potom tomto rozhodovacím faktoru snaží sledovat směr největšího poklesu S.
- **Následování ostatních agentů** – D
    - Druhou položkou silového pole podlahy je pole D. Na rozdíl od S je toto pole dynamické a jeho hodnoty odrážejí intenzitu využití jednotlivých buněk. Toto pole informuje agenty o drahách ostatních agentů jako je tomu v modelech využívající nepřímou komunikaci pomocí stigmergie. Při opuštění buňky agent zanechá o jejím použití informaci v podobě zvýšení hodnoty D o definovanou hodnotu. Tato informace se v čase mění. V každém kroku simulace na všech buňkách s nenulovou hodnotou D dojde k tomu, že se tato hodnota s pravděpodobností $\delta \in [0, 1]$ o nějakou hodnotu sníží (rozpad) a s pravděpodobností $\alpha\ \delta \in [0, 1]$ dojde k částečné difuzi hodnoty D na okolní buňky.

- Jaký je použitý konkrétní rozhodovací model?
- Pokud je rozhodovací model řízený daty, odkud pocházejí tato data?
- Na jaké úrovni agregace byla tato data posbírána?

Individuální rozhodování: Subjektem rozhodnutí jsou všichni agenti, objektem rozhodnutí je směr pohybu během aktuálního simulačního kroku. Rozhodování je víceúrovňové:

- V první úrovni rozhodnutí agenti rozhodují o své ideální volbě směru pohybu.
- V druhé fázi rozhodnutí se musí vyřešit konflikty, kdy se více agentů v první úrovni rozhodování rozhodlo pro jednu buňku. Na buňku může vstoupit pouze jeden agent. Agenti se mohou také rozhodnout krok neprovádět (a na svůj „ideální" krok počkat).

Vnímání překážek a ostatních agentů – A_α: Agent dokáže vnímat jaké jsou překážky v jednotlivých směrech, kterými se může pohybovat, tj. rozezná, co se nachází na určitém počtu buněk v jednotlivých šesti směrech, počet takto rozeznatelných buněk je rozhled agenta – parametr označený jako r.

Pokud pro buňku, na které se nachází agent, jehož rozhodnutí vyšetřujeme označíme $f_{\alpha,i}$ jako „číslo obsazenosti" pro i-tou buňku v jednom možném směru pohybu α jako:

$$f_{\alpha,i} = \begin{cases} 1, & buňka\ C_{\alpha,i}\ je\ obsazená\ agentem \\ 0, & buňka\ C_{\alpha,i}\ je\ prázdná \end{cases} \qquad (29)$$
$$\alpha = 1,2,..,6; \qquad i \in \mathbb{N}; i \leq r$$

Můžeme spočítat ukazatel obsazenosti ve směru:

$$A_\alpha = (1 - \frac{1}{r}\left(r - r_\alpha + \sum_{i=1}^{min(r,r_\alpha)} f_{\alpha,i} \right) \qquad (30)$$

Kde r_α je vzdálenost od daného agenta k nejbližší překážce ve směru α. Hodnota A_α závisí na počtu ostatních agentů nacházejících se do vzdálenosti *r* na buňkách ve směru pohybu α od daného agenta a na vzdálenosti první pevná překážky (všechny buňky za překážkou zbývající do vzdálenosti rozhledu agenta *r* se započítávají jako překážka). $A_\alpha$nabývá hodnot z intervalu $[0, 1]$. Hodnota 1 znamená, že se ve výhledu agenta v daném směru až do vzdálenosti r nenachází žádný jiný agent či překážka naopak hodnota 0 znamená, že se agent daným směrem nemůže pohybovat vůbec, jiný agent či překážka je hned na sousední buňce.

Setrvačnost pohybu – K_I: K rozhodnutí o směru pohybu přispívá určitý setrvačný efekt agentů. Pokud se agent pohybuje určitým směrem, rád by v tom daném směru pokračoval, pokud ho směr nebude něco nutit měnit. Agent si tedy musí pamatovat svůj směr pohybu a zohlednit ho při rozhodování v dalším kroku. Pro agenta a zavedeme pro aktuální krok t ukazatel setrvačnosti $I_{\alpha,t}$, který se bude

podílet na rozhodnutí o preferovaném směru pohybu. Zvýhodněn bude pohyb ve stejném směru, v jakém byl proveden minulý krok agenta:

$$I_{a,t} = \begin{cases} 0, & \alpha_{a,t} \neq \alpha_{a,t-1} \\ > 0, & \alpha_{a,t} = \alpha_{a,t-1} \end{cases} \quad (31)$$

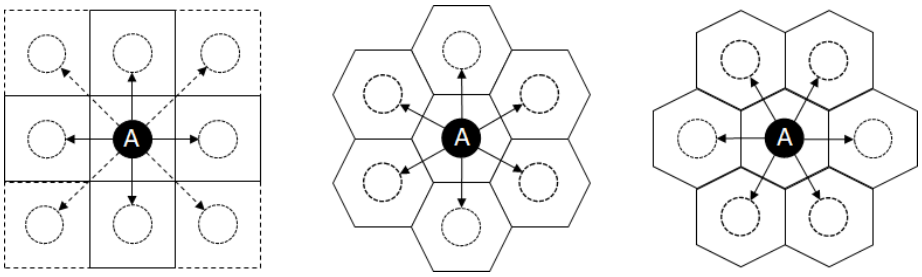
Prostorový dosah vnímání: Rozhled agenta r. Agenti získávají explicitně informace o poloze ostatních agentů do vzdálenosti svého rozhledu r. Agenti znají informace o prostředí (polohy překážek, zdí a východu.

Skupinové chování: díky použití dynamické složky silového pole podlahy D. V současné verzi modelu agenti žádná explicitní či cílená uskupení netvoří. Jde o možné rozšíření modelu o další složku kolektivního rozhodování, při kterém budou agenti snažit udržet v kontaktu s některými ostatními agenty záměrně (např. členové jedné rodiny). Modelování skupin chodců se věnuje např. práce (Lu at al., 2014) a také aktuální přehledová studie (Lu at al., 2017).

6.2.2 Implementace

Samotný model v NetLogu má dvě nezávislé části se stejnou funkcionalitou. První část využívá pouze prostředků samotného NetLoga, druhá využívá paralelní zpracování pomocí NL2OCL lze tak porovnat rychlost obou přístupů. Výsledky tohoto porovnání jsou k dispozici v kapitole 6.2.3 - Experimenty.

Příprava prostředí a populace agentů: Hexagonální síť buněk je vytvořena pomocí speciálního typu agentů – buňka (Cell). Tyto buňky jsou vytvořeny na všech buňkách původní čtvercové buněčné sítě – obr. 32(a). Základní Hexagonální síť je vytvořena ze sítě čtvercové tak, že nejprve jsou agenti typu cell umístěny na všechny buňky čtvercové sítě. Následně jsou agenti typu cells, které leží na sudých sloupcích (xcor mod 2 = 0), posunuty o vzdálenost rovnou půlce délky hrany buňky ve směru osy y. Agenti typu cells jsou potom uspořádáni do hexagonální sítě. Takto se používá hexagonální síť v knihovních modelech NetLogo – obr. 32(b). V modelech chodců na bázi buněčných automatů, které využívají tuto konstrukci je třeba rozhodnout, jaká bude orientace šestiúhelníků tvořených jednotlivými agenty typu cells. Další možností je pootočení směru šestiúhelníkových buněk – obr. 32(c).

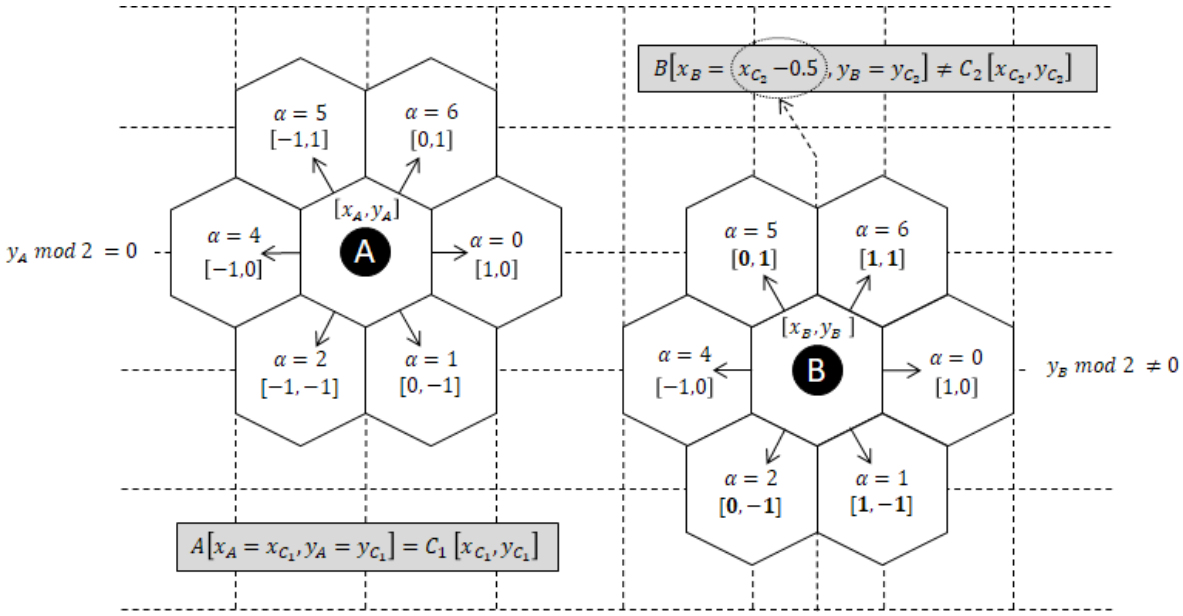


(a) (b) (c)

Obrázek 32 – různá uspořádání buněčných sítí, (a) – čtvercové, (b) – klasické hexagonální, (c) – pootočené hexagonální, (zdroj autor).

Bylo ukázáno, že hexagonální síť, ve které přímý směr pohybu mezi jednotlivými buňkami souhlasí s majoritním směrem pohybu chodců vykazuje realističtější výsledky simulací (Lee at al., 2010). V modelu EVAKUACE je také použita rotovaná hexagonální síť, viz obr. 32(c).

V této síti je zapotřebí nově definovat pojmy sousedních směrů a sousedních buněk. Existuje celkem 6 sousedních směrů, které označíme $\alpha \in \{0, 1, 2, 3, 4, 5\}$. Buňce hexagonální sítě, která je umístěná na souřadnicích buňky čtvercové sítě C [x, y], náleží sousední buňky hexagonální sítě, odpovídají buňkám čtvercové sítě podle toho, zda je y-nová souřadnice výchozí buňky sudé nebo liché číslo – viz obr. 33.



Obrázek 33 - Hexagonální síť vybudovaná na čtvercové síti a ofsety sousedních buněk, zdroj: Autor

Existují tedy dvě sady ofsetů souřadnic sousedních buněk lišících se pro směry 1, 2 a 4, 5 – můžeme je zapsat jako šestice uspořádaných dvojic:

Tabulka 9 - Ofsety souřadnic sousedních buněk v rotované hexagonální buněčné síti, zdroj: Autor

$Směr\ \alpha$:	0	1	2	3	4	5	Pro případ
$(dx, dy)_\alpha$	(1, 0),	**(0, -1),**	**(-1, -1),**	(-1, 0),	**(-1, 1),**	**(0, 1)**	$y_C\ mod\ 2 = 0$
	(1, 0),	**(1, -1),**	**(0, -1),**	(-1, 0),	**(0, 1),**	**(1, 1)**	$y_C\ mod\ 2 \neq 0$

Počítání s těmito ofsety je důležité nejenom při hledání přímých sousedních buněk, ale také při počítání ukazatele A_α, který určuje obsazenost okolí buňky C ve směru α podle vztahu (30). V modelu jsou tyto množiny ofsetů asociovány s jednotlivými agenty typu buňka (Cell) tak, aby v každém výpočetním kroku bylo možné rychle určit hexagonální okolí aktuální buňky.

Ve výše zmíněných knihovních modelech NetLoga, ve kterých se hexagonální síť používá, jsou ofsety souřadnic sousedních buněk uloženy vždy jako kopie celého pole ofsetů, které jsou uloženy jako atributu buňky. V tomto modelu je to provedeno odlišným způsobem. Protože je určen pro rozsáhlé simulace a pro paralelní zpracování, je třeba způsob uložení dat optimalizovat. Pole ofsetů sousedních buněk v modelu existují jako globální paměťové struktury, přičemž u jednotlivých buněk je zaznamenám pouze odkaz na jedno ze dvou polí, které se pro výpočet sousedů použije. Pro NetLogo prostředí s rozměry 500 x 500 buněk, se kterým se dále v tomto modelu experimentuje, to znamená značnou úsporu, přičemž nejde ani tak o samotnou paměť, jako o optimalizaci datových přenosů mezi NetLogem a platformou OpenCL.

Inicializace: Vytvoření zdí - buňky na okraji prostředí modelu jsou označeny jako zdi, stejně mohou být označeny i jiné buňky uvnitř prostředí, reprezentují potom překážky.

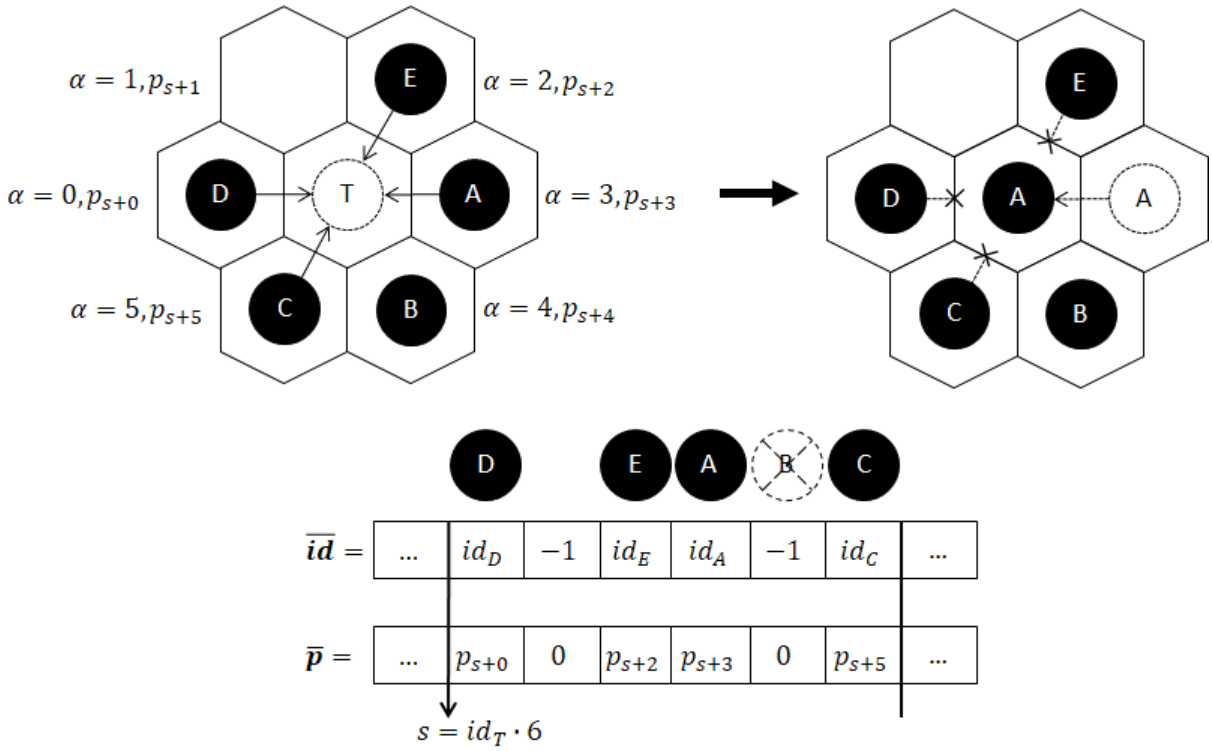
Vytvoření východu – východ je tvořen specifikovaným počtem buněk pravé zdi prostředí. To, jestli je buňka zdí nebo dveřmi nebo zda se na ní nachází nějaký agent či je neobsazená, tyto informace model při iniciaci uloží do uživatelského atributu buňky modelu. Tento atribut je později použit, pro vytvoření paměťového bufferu, který je v rámci paralelního zpracování simulačního kroku modelu poskytnut jako argument výpočetnímu kernelu.

Vytvoření populace agentů: Agenti jsou vytvořeni v požadovaném počtu a rozmístěni náhodně v prostředí modelu, jsou nastaveny iniciální hodnoty atributů agentů.

Příprava paměťových objektů pro argumenty kernelu: Pro stavové veličiny agentů a buněk, které jsou používány výpočetním kernelem, jsou vytvořeny paměťové buffery pomocí registračních funkcí programového rozšíření NL2OCL. Jsou také specifikovány požadavky na automatickou synchronizaci (NetLogo -> NL2OCL -> OpenCL a zpět) pro atributy, které se mají po provedeném paralelním výpočtu aktualizovat.

Proces rozhodování o vítězném agentovi, který obsadí v rámci simulačního kroku buňku, o kterou má zájem více agentů: Pokud je pro více agentů jedna buňka určena jako ideální pro provedení jejich dalšího kroku, je nutné rozhodnout o tom, který z agentů na tuto buňku skutečně vstoupí. Podle popisu chování agentů probíhá toto rozhodnutí ve dvou fázích – v první fázi je uplatněn parametr MI, který určuje pravděpodobnost, se kterou se konfliktní situace vyřeší jednoduše tak, že na buňku nevstoupí žádný z konkurujících si agentů. Hodnota tohoto parametru pro konkrétní buňky se mění, čím blíže se buňka nachází východu, tím je pravděpodobnost soupeření o danou buňky větší čili pravděpodobnost neřešení konfliktu M_i je menší.

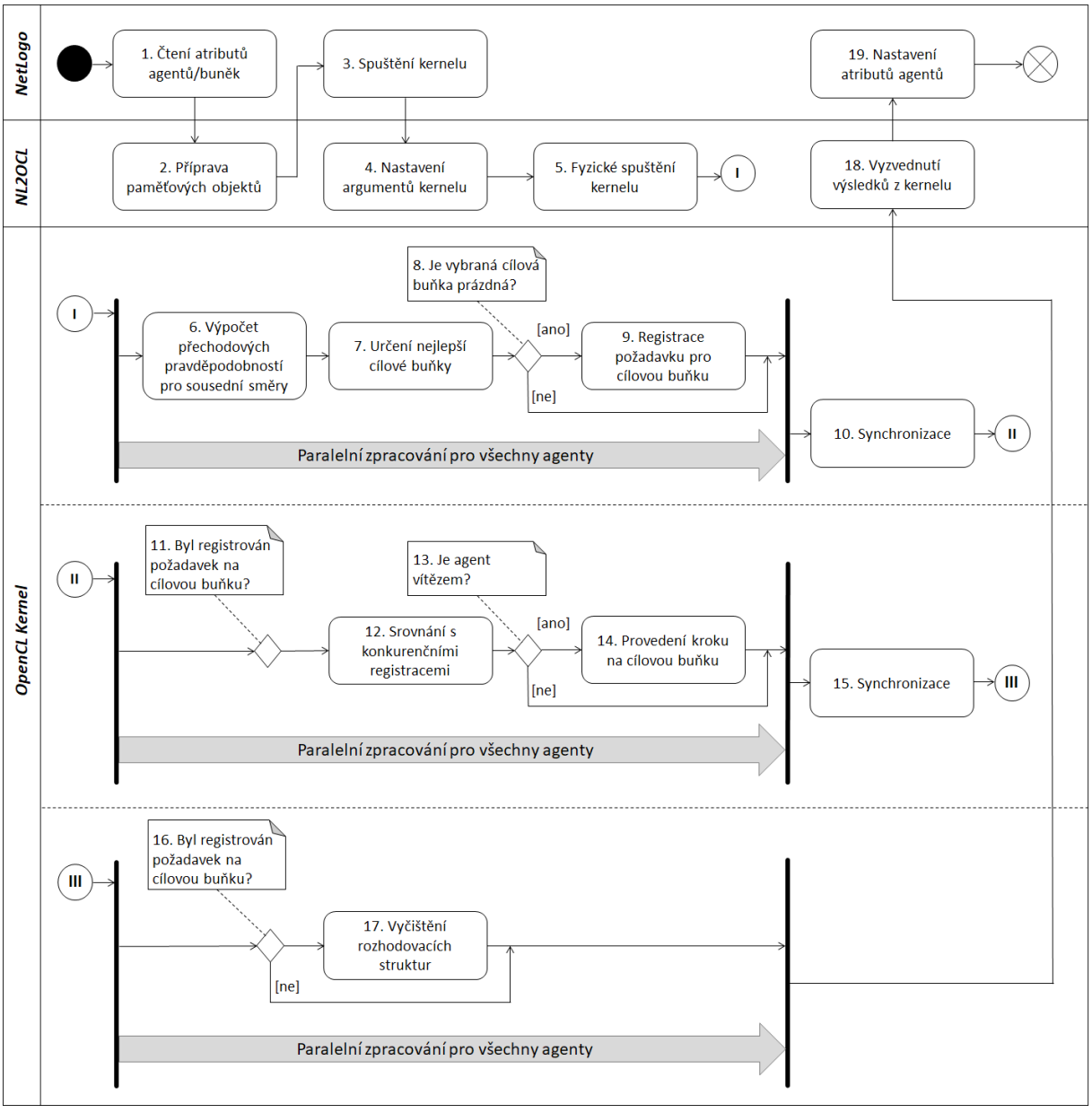
Pokud je rozhodnuto, že na buňku některý z agentů vstoupí (pravděpodobnost tohoto jevu je 1 - Mi), rozhodnutí o tom, který to bude je druhou fází rozhodování. V této druhé fázi mají všichni agenti buďto stejnou šanci na úspěch nebo jsou někteří agenti preferováni díky svým vlastnostem např. díky fyzickým předpokladům. Pokud jsou pravděpodobnosti úspěchu jednotlivých agentů shodné, jedná se vlastně o implementaci operace **one-of** tak, jak je dostupná v NetLogu, která vybírá náhodně jednoho z množiny agentů. Při paralelním zpracování je situace komplikovanější. Agenti provádějí svoje individuální vnímání okolí a rozhodování odděleně. Toto oddělené zpracování je prováděno paralelně. Kód kernelu zodpovědného za simulační krok se tedy nikdy neoctne v situaci, kdy by se pro jednu buňku rozhodovalo o tom, kdo z agentů, kteří se o ni ucházejí, vstoupí.



Obrázek 34 – rozhodování typu "one-of" v konfliktní situaci, zdroj: Autor

Na obr. 3 je ilustrován rozhodovací proces konkurenčního přístupu k jedné buňce více agenty. V okolí cílové buňky T (target) se nachází celkem pět agentů, z nich čtyři usilují o vstup na toto buňku. Ve směru $\alpha = 1$ žádný agent není. Agent B o vstup na buňku neusiluje. Rozhodování se tedy účastní agenti A, C, D a E. Každý může vstoupit na buňku ze svého směru. Směry jsou označeny po řadě: $\alpha = 3, 5, 0\ a\ 2$ (jde o směry z pohledu jednotlivých agentů, ne z pohledu buňky T). každý agent provede registraci svého požadavku na vstup na buňku T do speciálních datových struktur – pole $\overline{id}, \overline{p}$. Tato pole mají kapacitu na pokrytí celého prostředí modelu a každému možnému pohybu všech agentů náleží právě jeden člen pole. Na každou buňku lze vstoupit pouze ze 6 směrů, v každém směru se může nacházet pouze jeden agent. Takto je zabezpečeno, že čtení a zápis do těchto polí nemusí být při fázi rozhodování

agentů nijak synchronizován. Jedná se o lock-free sdílenou datovou strukturu. Agenti A, C, D a E provedou svoji registraci tak, že zaznamenají svůj identifikátor a také vygenerovanou pravděpodobnost jejich úspěchu na příslušné pozice v polích. Prvky náležející směrům, ze kterých se na buňku nechystá vstoupit žádný agent, zůstanou nastaveny na iniciálních hodnotách. Poté, co všichni agenti zaznamenají své hodnoty, může dojít k rozhodnutí o vítězi. Vítěz je určen podle pravděpodobností úspěchu. Podstatné je to, že každý agent může poznat, zda je vítěz či nikoli nezávisle, přečtením všech šesti prvků příslušného úseku pole. To je mechanizmus, jak lze o vítězi rozhodnout při paralelním zpracování, kdy jednotliví agenti nevědí o tom, v jaké části výpočtu se právě nacházejí.



Obr. 35 - UML diagram aktivit paralelního zpracování simulačního kroku evakuačního modelu, zdroj: Autor

Obr. 35 zobrazuje diagram aktivit pro paralelní zpracování jednoho simulačního kroku v evakuačním modelu. Za různé aktivity jsou zodpovědné různé části heterogenního výpočetního systému NetLogo-

NL2OCL-OpenCL, v UML notaci je to zachyceno pomocí tzv. „plaveckých drah“ pro jednotlivé části. V tabulce 10, která je uvedena níže, jsou jednotlivé aktivity popsány podrobně. Kromě popisu aktivity jsou popsány také výstupy, které jednotlivé aktivity produkují.

OpenCL kernel zpracovává vlastní výpočet pro mnoho agentů najednou v paralelně běžících vláknech, v OpenCL terminologii pracovních položkách (work item). Tato vlákna jsou organizována v OpenCL do tzv. pracovních skupin (work groups). V diagramu aktivit na obr. 35 jsou detailně rozepsány aktivity týkající se jednoho agenta, které jsou zpracované jedním výpočetním vláknem.

Paralelní výpočet musí být na určitých místech synchronizován. Tato synchronizace je nutná právě kvůli paralelní povaze zpracování výpočtu. Pro jednotlivé agenty jsou ve fázi I připravena data, která jsou následně ve fázi II využita i ostatními agenty pro jejich rozhodování. Spuštění fáze II tedy musí počkat na dokončení fáze I pro všechny agenty, tj. na dokončení fáze I všemi výpočetními vlákny. Další synchronizační krok je mezi fázemi II a III. Fáze III provádí čištění datových struktur, které obsahují data používaná pro rozhodování ve fázi II. Toto čištění probíhá proto, aby se kernel připravil na svůj další běh pro výpočet v dalším simulačním kroku modelu. Kdyby fáze III nepočkala na dokončení fáze II všemi vlákny, nastala by situace, že některá vlákna by už svoji část dat určenou pro rozhodování smazala a tato data by pak již nebyla k dispozici pro ostatní agenty, kteří teprve svoje rozhodování dokončují ve fázi II. Specifikace požadavků na synchronizaci při paralelním zpracování je velmi důležitá a představuje zásadní bod návrhu paralelního algoritmu zpracování výpočtu.

Tab. 10 – popis aktivit paralelního zpracování simulačního kroku, zdroj: Autor

#	Popis aktivity	Výstup aktivity
1	Čtení atributů agentů/buněk – v NetLogu se pomocí registračních funkcí (např. RegisterIntBufferHolderFromAS) z množiny agentů a z buněk pro vybrané atributy připraví datové struktury vhodné pro vytvoření paměťových objektů, které budou nastaveny jako argumenty OpenCL kernelu. Jako parametry jsou poskytnuty informace o atributech, datových typech, identifikaci kernelu a pořadí jeho argumentů a o požadavcích na synchronizaci pro jednotlivé argumenty.	Objekty v NL2OCL spravující datové struktury, které umožňují synchronizovat atributy agentů a buněk s paměťovými buffery
2	Příprava paměťových objektů – v NL2OCL se připraví paměťové buffery odpovídajících datových typů a zkopírují se do nich hodnoty specifikovaných atributů agentů a buněk.	Paměťové buffery
3	Iniciace spuštění kernelu – v NetLogu se iniciuje spuštění kernelu pomocí funkce RunKernel, jako parametry jsou poskytnuty informace o rozsahu a dimenzích úlohy (NDRange a globální/lokální velikost pracovní skupiny)	Parametry o rozsahu a dimenzích paralelní úlohy pro spuštění kernelu

4	Nastavení argumentů kernelu – NL2OCL nastaví argumenty do připraveného kernelu v pořadí specifikovaném při registraci paměťových bufferů pro jednotlivé atributy agentů a buněk.	Kernel s nastavenými argumenty připravený ke spuštění
5	Fyzické spuštění kernelu – připravený kernel s nastavenými argumenty se pošle do fronty úloh kontextu OpenCL aplikace ke zpracování.	Běžící OpenCL kernel
Zpracování kernelu – fáze I Tato fáze představuje implementaci toho, jak agent vnímá svoje okolí a jak se rozhoduje o výběru svého nejvhodnějšího dalšího kroku, tj. nejvhodnější sousední buňky, na kterou by se agent rád přesunul.		
6	Výpočet přechodových pravděpodobností pro sousední směry. Vypočítají se ukazatelé obsazenosti podle vztahu (30) pro všech šest okolních směrů a určí se přechodové pravděpodobnosti na základě hodnot statického a dynamického silového pole podlahy (parametry S, D příslušné sousední buňky) a dalších ukazatelů. Pravděpodobnosti se normují tak, aby jejich součet byl roven 1.	Normované přechodové pravděpodobnosti pro jednotlivé sousední směry
7	Určení nejlepší cílové buňky – rozhodnutí agenta o nejlepší cílové buňce je provedeno na základě přechodových pravděpodobností (ne jako maximum, ale podle distribuce pravděpodobnosti)	Zvolená cílová buňka (směr)
8	Je vybraná cílová buňka prázdná? – kontrola, zda vybraná buňka je prázdná a je možné na ni vstoupit. Pokud ano, následuje krok 9, pokud ne, fáze I výpočtu pro tohoto agenta končí	Rozhodnutí, zda je možné na buňku vstoupit
9	Registrace požadavku pro cílovou buňku – naplnění rozhodovací datové struktury pro řešení konkurenčního přístupu na jednotlivé buňky. Agent zaznamená svoji pravděpodobnost úspěchu, která se pro něj vygeneruje, na specifické místo rozhodovací struktury jako svoji registraci toho, že o tuto buňku usiluje. Přesné místo v rozhodovací datové struktuře, kam agent svoji pravděpodobnost úspěchu zaznamená je odvozeno od toho, o jakou buňku se jedná (od jejích souřadnic) a od toho, z jakého směru se agent na buňku chystá vstoupit (směr 0-5).	Zaregistrovaný požadavek vstupu na buňku
10	Synchronizace – globální synchronizace všech výpočetních vláken. Všichni agenti musí dokončit fázi I výpočtu.	Všichni agenti dokončili fázi I výpočtu
Zpracování kernelu – fáze II Tato fáze představuje vyřešení konfliktů mezi agenty, kteří usilují o vstup na stejnou buňku. Pro řešení konfliktu je použita speciální lock-free datová struktura.		
11	Byl registrován požadavek na cílovou buňku? – pokud agent během aktivity č. 9 nezaregistroval požadavek vstupu na nějakou cílovou buňku, výpočet pro něj v podstatě končí.	Rozhodnutí o tom, zda agent pokračuje ve výpočtu a zda usiluje o vstup na nějakou buňku
12	Srovnání s konkurenčními registracemi – agent prohlédne registrace ostatních agentů pokoušejících vstoupit na stejnou buňku z ostatních směrů. Pokud je agent jediným agentem	Srovnání pravděpodobnosti úspěchu agenta s ostatními agenty

	usilujícím o tuto buňku, je pravděpodobnost jeho úspěchu rovna 1.	usilujícími o stejnou buňku
13	Je agent vítězem? – výsledkem srovnání s konkurenčními agenty je rozhodnutí o tom, zda je agent vítěz či nikoli.	Informace o tom, zda agent zvítězil v boji o buňku či nikoli.
14	Provedení kroku na cílovou buňku – je proveden krok agenta, zatím jen datově, to znamená, že paměťové buffery reprezentující informace o poloze agenta a obsahu jednotlivých buněk hexagonální sítě jsou aktualizovány. K realizaci fyzického kroku v NetLogo modelu dojde až po zasynchronizování příslušného paměťového bufferu s odpovídajícími atributy reprezentujícími polohu daného agenta (atributy xcor a ycor)	Aktualizované paměťové buffery zaznamenávající provedený krok agenta.
15	Synchronizace – globální synchronizace všech výpočetních vláken. Všichni agenti musí dokončit fázi II výpočtu.	Všichni agenti dokončili fázi II výpočtu
Zpracování kernelu – fáze III Během této fáze se vyčistí datová struktura pro rozhodování o konkurenčním přístup agentů ke stejným buňkám.		
16	Byl registrován požadavek na cílovou buňku? – čištění se provádí jen pro agenty, kteří nějakou registraci do rozhodovaní datové struktury v aktivitě č. 9 provedli. Pro ostatní agenty je to zbytečné.	Rozhodnutí, zda agent provádí čištění či nikoli
17	Vyčištění rozhodovacích struktur – agent zaznamená na místo rozhodovací struktury, kam během aktivity č. 9 zaznamenal svoji pravděpodobnost úspěchu, hodnotu -1. Tím je místo v datové struktuře odpovídající vstupu na danou buňku ze směru, ze kterého na ní agent vstoupil, připraveno na použití v dalším simulačním kroku.	Aktualizovaná rozhodovací datová struktura (jeden konkrétní prvek změněný na hodnotu -1)
Aktivitou 17 skončil výpočet kernelu. Pro všechny agenty je kernel na konci výpočtu. Na dokončení výpočtu všech výpočetních vláken kernelu se počká automaticky. Následuje výběr vypočítaných hodnot z kernelu a jejich použití pro aktualizaci situace v modelu.		
18	Vyzvednutí výsledků z kernelu – tato aktivita vyzvedne vypočítaná data z OpenCL zařízení do host aplikace, odkud je možné je přenést do modelu.	Aktualizované datové struktury držící hodnoty atributů
19	Nastavení atributů agentů – realizace skutečného pohybu agentů (kteří vykonali krok na nějakou buňku) na základě aktualizovaných atributů jejich souřadnic.	Nastavené vypočítané hodnoty do atributů modelu

6.2.3 Experimenty

S modelem evakuace byly provedeny dva experimenty.

Experiment 1

První experiment byl zaměřen na provedení celkové simulace pro velkou populaci agentů ve velkém prostředí. Cílem experimentu bylo ověřit, že implementace vnímání, rozhodování a pohybu agentů

pomocí paralelního zpracování bude při rozsáhlé simulaci vykazovat očekávané chování, které bylo ověřeno při simulaci s menší populací, která byla provedena prostředky NetLoga. Dále měl experiment ověřit, že běh takto velké simulace je možný v reálném čase, a že doba provedení celé simulace od vytvoření populace agentů, až po úplné uvolnění prostředí, je akceptovatelná pro použití modelu pro provádění opakovaných simulace pro různé evakuační scénáře.

- *Prostředí modelu*

  500 x 500 buněk modelu – hrana jedné čtvercové buňky odpovídá 40 cm, potom rozsah reálného simulovaného prostředí je 200 x 200 m. Šířka východu byla zvolena 40 buněk to odpovídá celkové šířce reálného východu 10 m.
- *Agenti*

  Počáteční populace byla 50 000 agentů rozmístěných náhodně do prostředí modelu, vzdálenost, na kterou agent rozeznává okolní prostřední (parametr r) byla nastavena na 10 buněk.
- *Průběh simulace*

  Simulace byla spuštěna, jednotlivé kroky se opakovaly automaticky, výsledky vypočítané paralelně pomocí NL2OCL se automaticky synchronizovaly s grafickou reprezentací agentů v NetLogo, průběh bylo možné sledovat v reálném čase. Průběh simulace je zachycen na obr. 36. jsou zde zachyceny stavy simulace po 1 500 krocích (legenda jednotlivých obrázků je: počet provedených simulačních kroků / počet zbývajících agentů).
- *Celkový počet simulačních kroků*

  Pro odchod všech agentů mimo prostor prostředí modelu bylo třeba celkem 20 147 simulačních kroků.
- *Zrychlení při použití NL2OCL oproti NetLogu*

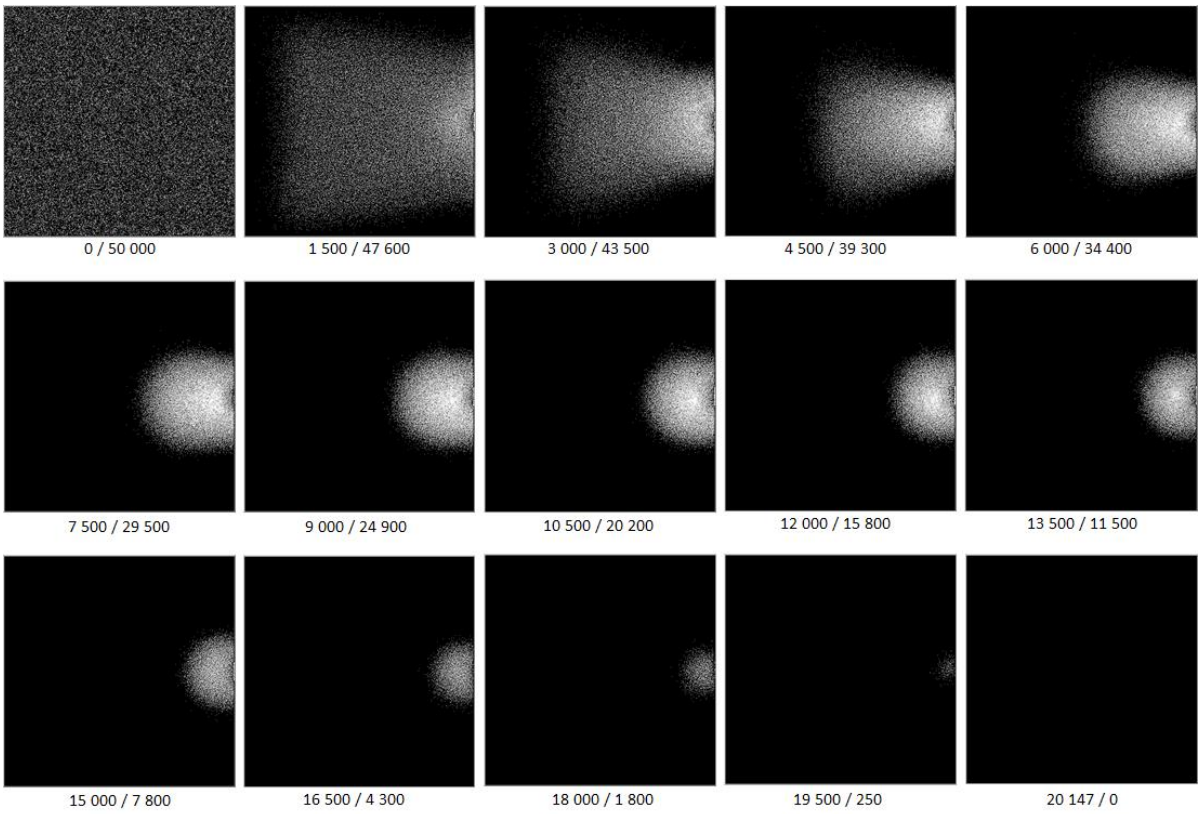
  Při počáteční velikosti populace agentů (50 000) byla průměrná doba trvání simulačního kroku při výpočtech prováděných pouze prostředky NetLoga průměrně 7,3 s. Při použití programového rozšíření NL2OCL trval průměrný simulační krok jen 0.06 s, to znamená **celkové zrychlení simulace cca 120krát**.
- *Doba trvání celé simulace*

  Simulace trvala celkem 690 s, průměrná doba trvání jednoho simulačního kroku byla 0,034 s.

Vyhodnocení experimentu 1

Simulace při použití NL2OCL probíhala 120krát rychleji než při výpočtech prováděných pouze prostředky systému NetLogo. Tento výsledek ukazuje, že jak návrh způsobu paralelizace, tak provedení vlastní implementace paralelních výpočtů (konstrukce OpenCL kernelu) byly provedeny efektivním způsobem. Tento výsledek také demonstruje technické možnosti použití programového rozšíření NL2OCL na modelech typu buněčný automat s výpočty na základě rozšířeného okolí agentů

(s komplexnějším vnímání okolí a složitějším procesem rozhodování). Simulace, která nativně v NetLogu probíhala rychlostí řádově jeden simulační krok za jednotky sekund, může být pomocí paralelizace prostřednictvím NL2OCL provedena rychlostí řádově stokrát větší. To představuje dostatečný potenciál pro široké spektrum podobných simulací s komplexnějšími evakuačními scénáři, složitějšími vzorci chování a rozhodování agentů, v rozsáhlejším prostředím a s většími populacemi.



Obr. 36 – Průběh simulace prostředí modelu 500x500, 50 000 agentů, simulační krok / počet agentů, zdroj: Autor

Experiment 2

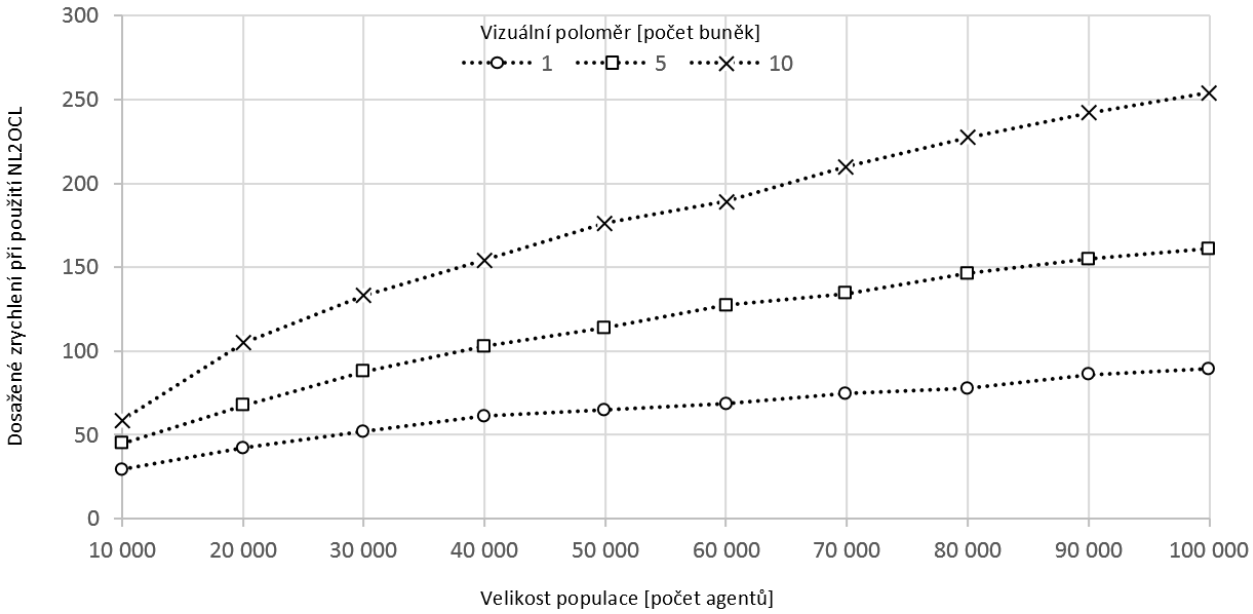
Porovnání dosažených zrychlení pro různé velikosti populace agentů a pro různé vizuální poloměry agenta r (vizuální poloměr je vzdálenost, na kterou agent dohlédne, tj. počet buněk v jednom směru, jejichž obsah dokáže agent zahrnout do svého rozhodování o nejlepším možném směru svého dalšího kroku).

Postup experimentu – Model byl nastaven tak, aby samotné prostředí NetLogo pracovalo co nejrychleji, tzn. kontrolní prvek umožňující měnit rychlost simulace byl nastaven na maximum. Dále bylo vypnuto automatické aktualizování grafické informace v prostředí modelu. V tomto experimentu šlo o porovnání maximálních možných výpočetních rychlostí, kdy ukazatelem byla doba trvání výpočtů potřebných pro uskutečnění jednoho simulačního kroku, vykreslování aktualizací by tyto časy zkreslovalo.

Iniciální parametry modelu – Postupně byl nastaven vizuální poloměr r na hodnoty 1, 5 a 10 a pro každý vizuální poloměr byly nastaveny počáteční populace 10 000 až 100 000 agentů. To znamená celkem 30 iniciálních stavů simulace. Pro každý iniciální stav byla simulace spuštěna ve dvou variantách. Při první variantě provádělo výpočty pouze NetLogo, ve druhé variantě byly výpočty prováděny paralelně pomocí NL2OCL. Byla měřena průměrná doba trvání jednoho simulačního kroku. Výsledky experimentu jsou zobrazeny v tab. 11 a v grafu na obr. 37.

Tabulka 11 – přehled dosažených zrychlení simulace v modelu EVAKUACE, pro rázné populace agentů a pro měnící se vizuální poloměr agenta r, zdroj: Autor

R			Počet Agentů									
			10 000	**20 000**	**30 000**	**40 000**	**50 000**	**60 000**	**70 000**	**80 000**	**90 000**	**100 000**
1	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	0,855	1,529	2,253	3,173	3,885	4,809	5,754	6,621	7,919	8,431
		NL2OCL	0,029	0,036	0,043	0,052	0,060	0,070	0,077	0,085	0,092	0,094
	Zrychlení simulačního kroku		**29**	**42**	**52**	**61**	**65**	**69**	**75**	**78**	**86**	**90**
5	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	1,389	2,625	3,953	5,468	6,962	8,659	10,467	11,865	14,121	15,293
		NL2OCL	0,031	0,039	0,045	0,053	0,061	0,068	0,078	0,081	0,091	0,095
	Zrychlení simulačního kroku		**45**	**67**	**88**	**103**	**114**	**127**	**134**	**146**	**155**	**161**
10	Průměrná doba trvání jednoho simulačního kroku [s]	NetLogo	1,997	3,873	5,993	8,175	10,418	12,881	15,757	18,620	21,769	24,421
		NL2OCL	0,034	0,037	0,045	0,053	0,059	0,068	0,075	0,082	0,090	0,096
	Zrychlení simulačního kroku		**59**	**105**	**133**	**154**	**177**	**189**	**210**	**227**	**242**	**254**



Obrázek 37 – Dosažená zrychlení v modelu EVAKUACE při použití programového rozšíření NL2OCL pro 10 000 až 100 000 agentů a vizuální poloměr 1, 5 a 10, zdroj: Autor

Vyhodnocení experimentu 2

Na základě hodnoty zrychlení v tabulce 11 a grafu na obr. 37 lze označit provedenou paralelizaci modelu EVAKUACE za řešení s dobrou škálovatelností. Zrychlení roste se zvětšující se složitostí simulační úlohy. Patrná nelinearita trendu růstu zrychlení je pravděpodobně způsobena dodatečnými výpočetními náklady na komunikaci, které rostou se zvětšující se populací nelineárně (vypočítané hodnoty paralelně pracující části se musejí přenést zpět do NetLoga). Zrychlení v řádu 200násobku základní rychlosti NetLoga dosažené při tomto experimentu pro populaci 100 000 agentů potvrzují technický potenciál NL2OCL, při správném způsobu využití, urychlit i značně rozsáhlé simulace.

6.3 Model OSÍDLENÍ – paralelizace modelu sítě keltských sídlišť

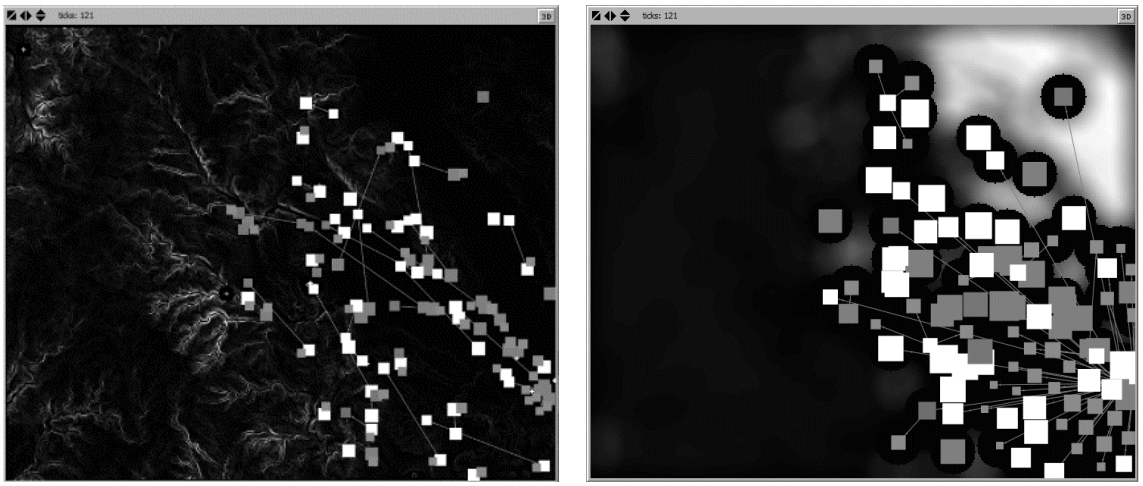
Tato kapitola se věnuje modelu popsanému v (Olševičová, Procházka a Danielisová 2015). Bude vysvětlena fáze návrhu modelu s ohledem na proces identifikace možností paralelizace výpočtů. Bude při tom využito sekvenčních diagramů, které byly v metodickém postupu pro paralelizaci multi-agentových modelů vytipovány jako vhodné nástroje pro fázi funkční dekompozice (viz kap. 4.2). Dále budou nastíněny implementační možnosti, jak by se paralelizace výpočtů pomocí prostředků NL2OCL v tomto modelu řešila. Vlastní implementace a experimenty popsány nebudou, neboť by přesahovaly kontext této práce.

6.3.1 Návrh

Cílem výzkumu bylo rekonstruovat růst sídelní sítě a studovat prostorové a funkční struktury této sítě. Konkrétně se týkal osídlení lokality Starého Hradiska v pozdní době železné (časové rozmezí od 4. st. př. Kr. do 1. st. po Kr.). Odpovídající výpočetní model jako své vstupy využívá jednak data z archeologických nálezů (mapa nalezišť a známých sídel) a dále GIS data o charakteristikách území, na kterém se síť budovaných sídel nachází. Výstupem simulace je situace v regionu reprezentovaném modelem po uplynutí požadovaného počtu let. Specificky je výstupem síť sídel a struktura jejich populace. Vizualizace možných konečných stavů simulace je zachycena na obr. 38.

V modelu jsou definovány následující druhy agentů:

- **Domácnosti** – elementární jednotka sdružující členy populace.
- **Sídla** – místa, kde domácnosti žijí, a která jim v rámci daného okolí poskytují zdroje obživy. Toto okolí vymezuje tzv. frakční rádius, který je daný maximálním časem, který jsou ochotni obyvatelé trávit přesunem za prací (pohybují se pěšky).
- **Uskupení sídel** – spojení sídel do větších prostorových celků. Rozrůstající sídla se nakonec spojují v tato uskupení.
- **Propojení** – speciální druh objektů propojujících jednotlivá sídla, reprezentují v modelu spojovací cesty (je k nim přistupováno jako k třídě agentů).



Obrázek 38 – Vizualizace konečného kroku simulace pro model OSÍDLENÍ, vlevo i vpravo je situace po 120 simulačních krocích pro dva různé simulační scénáře (odlišné parametry populační dynamiky), zdroj: Autor

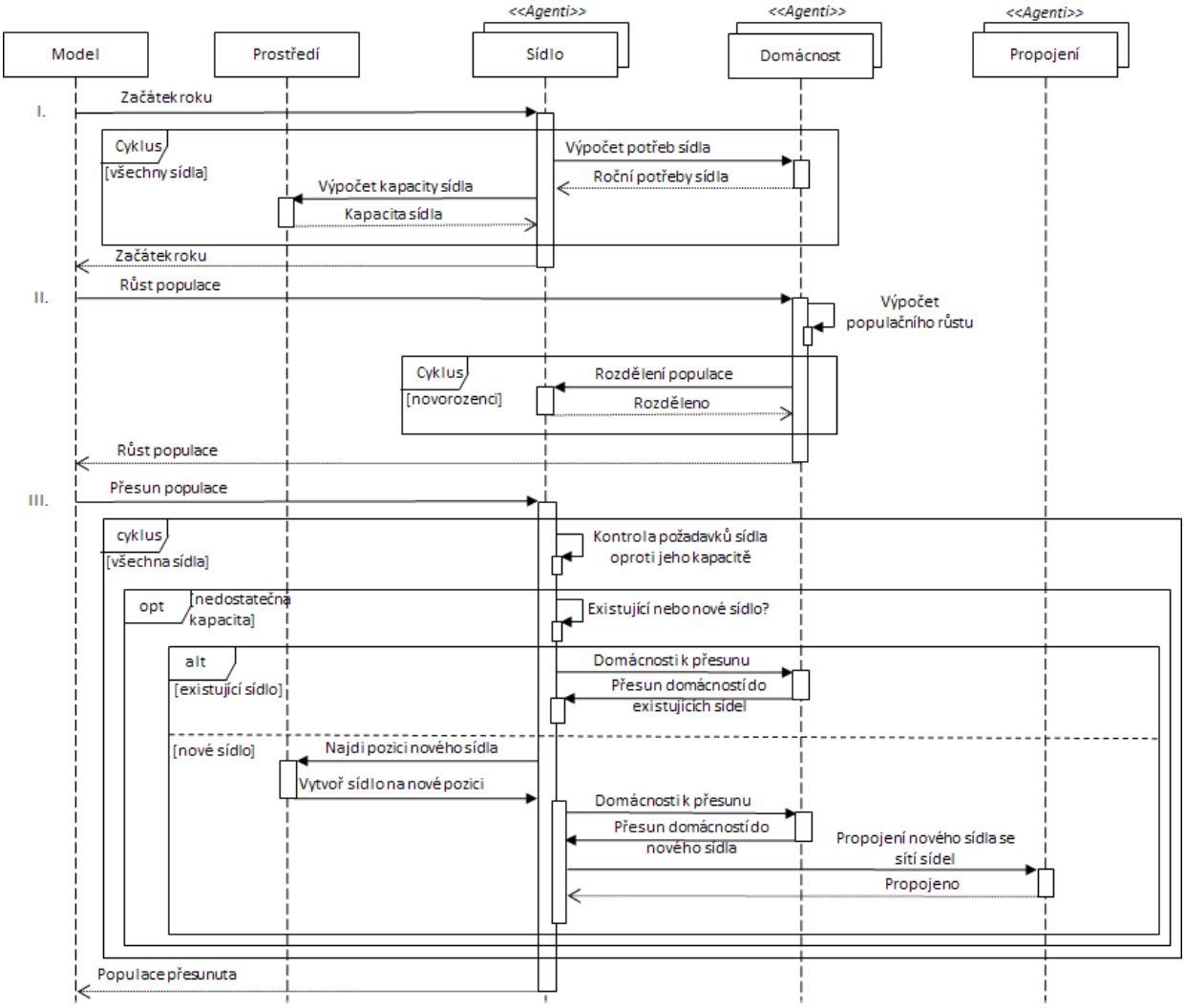
Prostředí modelu je tvořeno čtvercovou sítí buněk o rozměrech 360 x 300 buněk (velikost je dána vstupním souborem GIS dat). Jedna buňka představuje plochu 100 x 100 m^2. Celkové rozměry reálného prostředí jsou tedy 36 x 30 km^2.

Simulované období je 120 let (1 rok = 1 simulační krok). V rámci jednoho simulačního kroku je řešena sekvence různých vzorců chování různých tříd agentů. Funkční dekompozici lze provést na základě sekvenčního diagramu na obr. 38. Simulační krok se skládá ze třech nezávislých fází:

I. **Začátek roku** – aktivity spojené s výpočty pro daný simulační krok (jeden rok). Jde především o výpočet populační kapacity sídla, tj. kolik obyvatel je sídlo schopné vzhledem k dostupnosti zdrojů v okolí sídla pojmout (uživit)., a o výpočet ročních potřeb daného sídla, tj. spotřebu populace, která se aktuálně v tomto sídle nachází. Tato fáze reprezentuje sub-model zdrojů, produkce a spotřeby.

II. **Růst populace** – celková populace každý rok roste v závislosti na parametrech modelu. Noví členové populace navýší počet obyvatel v sídlech, ve kterých žijí domácnosti, do kterých se narodili. S novými členy populace rostou i nároky na spotřebu domácností a odpovídajících sídel. Tato fáze reprezentuje sub-model populační dynamiky modelu osídlení.

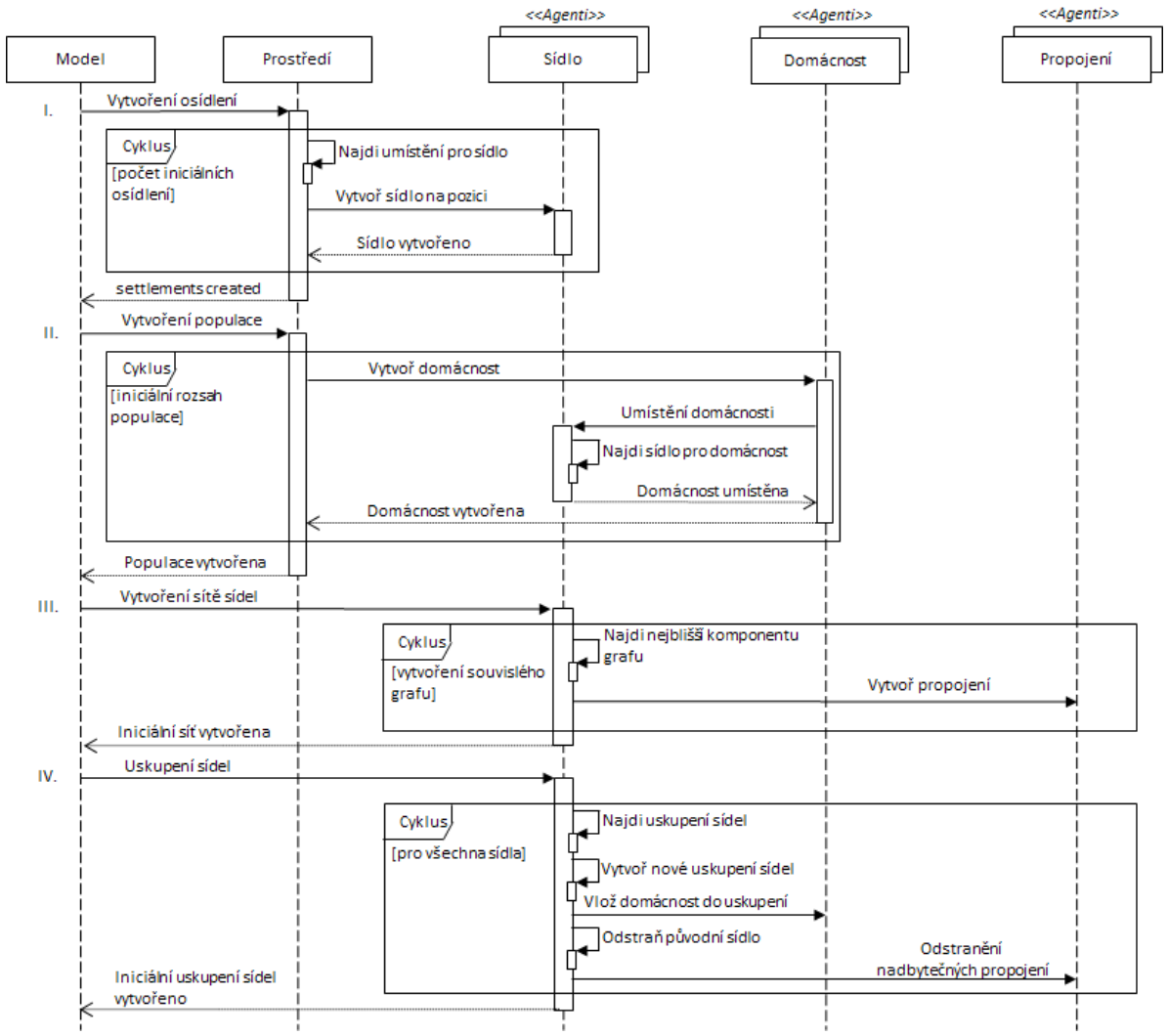
III. **Migrace** – zvětšující se populace je hlavní hnací silou rozšiřování sítě sídel. Pokud kapacita sídla pro danou populaci nestačí, někteří členové musejí odejít. Neodcházejí jednotlivci, ale celé domácnosti nebo jejich definované části. Buďto se stěhují do existujících sídel, která je mohou díky své ještě nenaplněné kapacitě ještě přijmout, nebo zakládají sídla nová. Umístění nového sídla vychází z prostorového uspořádání prostředí (tendencí stěhujících se domácností je stěhovat se jen do určité vzdálenosti od jejich původního sídla) a možností prostředí (kapacit určitého místa uživit určitou populaci).



Obrázek 39 – sekvenční diagram simulačního kroku modelu OSÍDLENÍ, zdroj: Autor

Nezanedbatelnou složkou modelu je jeho iniciační část, kterou je třeba také podrobit funkční dekompozici. V této části se skrývá značný potenciál k paralelnímu zpracování některých inicializačních výpočtů. K identifikaci konkrétních paralelizovatelných výpočtů může opět posloužit funkční dekompozice na základě sekvenčního diagramu na obr. 40. Toto jsou jednotlivé fáze inicializace:

I. **Vytvoření sídel** – parametry modelu určují, kolik iniciálních sídel se má vytvořit před začátkem samotné populace a jak budou tato sídla v prostředí modelu rozmístěna. Tyto parametry vycházejí z archeologických nálezů. Jedná se celkem o 97 sídel různých kategorií a kapacit. Pro rozmístění sídel je definovaná sada pravidel a scénářů. Vhodnost míst prostředí pro vytvoření sídla je také dána empirickými daty charakterizujícími prostředí (prostupnost terénu, přítomnost vodních toků, dostupnost zemědělsky využitelné půdy, dostupnost lesů).



Obrázek 40- sekvenční diagram vytvoření inicializačního stavu simulace modelu OSÍDLENÍ, zdroj: Autor

II. **Vytvoření iniciální populace** – iniciální populace je 1000 obyvatel. Věkový profil populace je parametrizován. Iniciální populace je rozdělená do jednotlivých domácností a domácnosti jsou umístěny do vytvořených sídel (rozmístění je také řízeno různými scénáři).

III. **Propojení sídel do sítě** – na základě dat o reálném prostředí jsou sídla propojen sítí cest. Iniciální síť je tvořena jako strom s minimální kostrou pomocí Jarníkova algoritmu (Primův).

III. **Seskupení sídel do větších celků** – sídla, která jsou v síti umístěna blízko sebe jsou propojena do jednoho většího sídla. Vzájemnou blízkost lze vymezit např. kontrolou překryvu frikčních poloměrů daných sídel. Takováto větší sídla potom mohou zastávat v modelu další, specializované role.

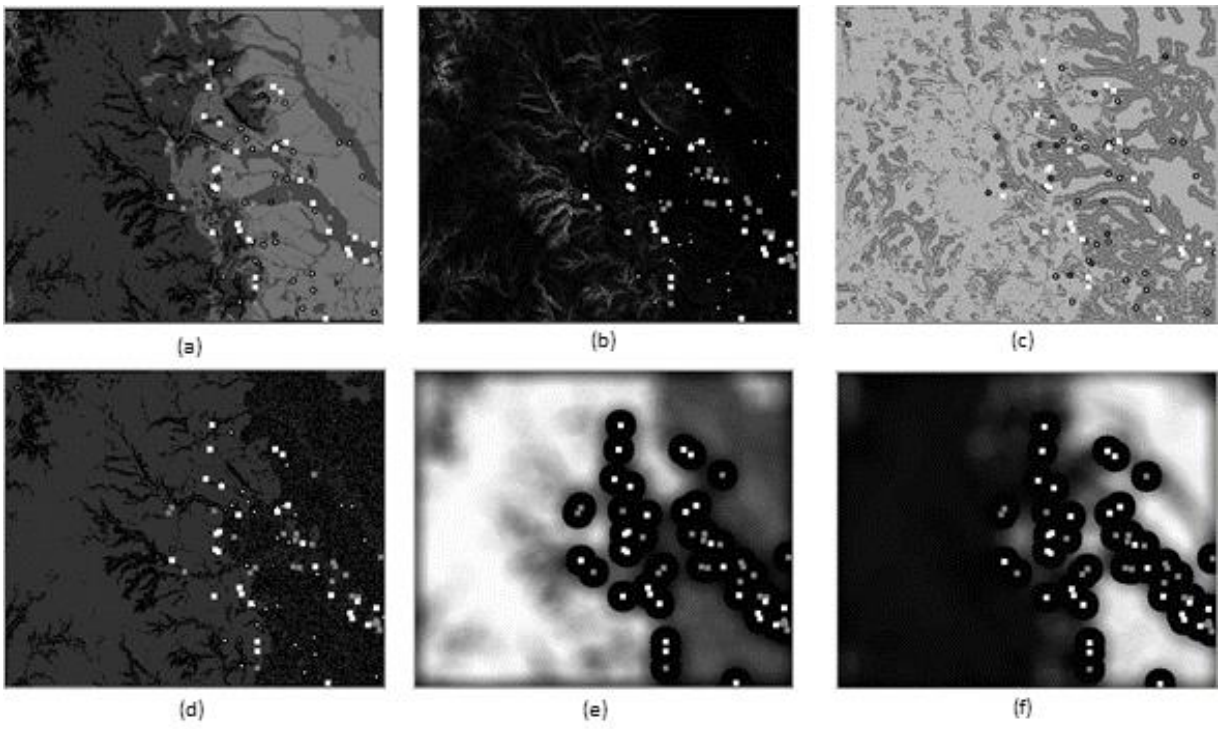
6.3.2 Možnosti paralelizace

V jednotlivých částech modelu popsaných v předchozí kapitole lze uvažovat o možnostech paralelizace zejména v případech, kdy probíhají výpočty opakovaně pro velké množství agentů nebo buněk prostředí. Tato místa je možné identifikovat právě pomocí sekvenčních diagramů (obr. 39 a 40) a to

v místech, kde je cyklické provádění operací jednou nebo více třídami agentů (UML stereotyp cyklus). Jako vhodné přicházejí v úvahu především tyto části:

- **Opakovaný výpočet potřeb a kapacit na začátku simulačního kroku** – pro každé sídlo se počítá na začátku simulačního kroku roční kapacita okolí sídla a roční potřeby obyvatel tohoto sídla. Do těchto výpočtů vstupují informace o prostředí (buňkách) nacházejících se v definované frikční vzdálenosti od sídla. Pro paralelizaci je vhodným kandidátem zejména úloha nalezení frikčního okolí sídla. Frikční okolí sídla není jeho topologické kruhové okolí. Jde o nepravidelné okolí dané prostupností terénu. Matematicky jde o úlohu hledání množiny vrcholů v grafu, jehož uzly tvoří buňky prostředí, hrany jsou spojnice mezi buňkami ohodnocené atributem prostupnosti terénu (přiřazeným jednotlivým buňkám na základě dat načtených z GIS zdrojových souborů) a požadavkem je dodržení maximální vzdálenosti každého uzlu této množiny od centrálního uzlu, kterým je buňka, na které se nachází agent reprezentující sídlo. Tento výpočet je možné implementovat jako OpenCL kernel a provádět paralelně pro všechny sídla.
- **Zakládání nových sídel pro rozrůstající se populaci** – pokud část populace sídla, které kapacitně přestane stačit zakládá nové sídlo, je třeba nalézt pro toto nové sídlo vhodné místo. Vhodnost místa je určena na základě několika pravidel. Jednak je stanovena maximální vzdálenost od původního sídla, přičemž tato vzdálenost se určuje pomocí parametru prostupnosti terénu. A dále musí nové místo poskytovat dostatečnou kapacitu (pro nově příchozí i pro budoucí rostoucí populaci daného sídla). Tato úloha je řešena pomocí kombinace grafových algoritmů a aplikace omezujících podmínek. Pro vytypovaná místa je ověřena kapacita prostředí. Jednak jde o místa odpovídající archeologickým nálezům a dále místa splňující podmínky vhodnosti (vhodnost je jedním z parametrů získaných na základě vstupních GIS dat odrážejících např. přítomnost vodních toků). Pro tato místa je proveden podobný výpočet jako při počítání roční kapacity již existujícího sídla. Hledání vhodného místa pro nové sídlo je vhodným kandidátem na paralelní zpracování. Paralelizovanou úlohou by v tomto případě byl výpočet vhodnosti konkrétního místa v prostředí k založení sídla.
- **Zpracování charakteristik prostředí** – charakteristiky prostředí jsou vypočítány na základě vstupních dat. Jedná se o data ve formátu GIS. Pro použití v modelu je třeba transformovat vstupní data do podoby atributů jednotlivých buněk na základě parametrů specifikovaných pomocí uživatelského rozhraní modelu. Vizualizace těchto atributů je na obr. 41. S měnícími se parametry modelu je nutné tyto atributy přepočítat. Z důvodu velikosti prostředí (360 x 300 buněk, tj. celkově 108 000 buněk) jsou tyto výpočty časově náročné. Současná verze modelu řeší zrychlení inicializace pomocí načítání již vypočítaných atributů (pro konkrétní sadu

parametrů modelu) ze souboru. Výpočty provedené paralelně pro všechny buňky by tento proces velmi urychlily. Při dostatečném zrychlení by se mohlo uvažovat o zakomponování možnosti změny parametrů prostředí do modelu za běhu simulace, což by otevřelo nové možnosti tvorby dalších simulačních scénářů zahrnujících změny prostředí v reálném čase (při zrychleních dosažených paralelizací modelů představených v kap. 6.1 a 6.2 je to reálné). Paralelním zpracování by se dalo uvažovat o zakomponování



Obrázek 41 – vizualizace parametrů prostředí v modelu OSÍDLENÍ pro atributy: (a) typ půdy, (b) prostupnost terénu – frikce, (c) kombinovaný ukazatel vhodnosti místa k založení sídla, (d) rozmístění lesů, (e) potenciál prostředí produkovat dřevo, (f) potenciál prostředí produkovat obilí, zdroj: Autor

6.4 Shrnutí poznatků z paralelizace vybraných modelů

V předchozích kapitolách bylo demonstrováno použití programového rozšíření NL2OCL na konkrétních modelech. Zde jsou uvedena některá zobecnění, která platí pro paralelizované NetLogo modely.

V NetLogu jsou jednotlivé příkazy pro agenty prováděny sekvenčně pro jednoho agenta po druhém. Pokud agent provede nějaké změny dat v modelu, dalšímu agentovi jsou tyto změny ihned dostupné. Lze říci, že data v modelu jsou synchronizována v každém kroku výpočtu. Po každém kroku výpočtu lze také jednoznačně říci, kteří agenti již tímto krokem prošli a kteří nikoli. Při paralelním zpracování toto jednoznačně říci nelze, jednotlivá vlákna zpracovávají výpočet pro velké množství agentů najednou a probíhají nezávisle. Při návrhu paralelního zpracování proto musejí být dobře identifikována místa

výpočtu, ve kterých je vyžadována nějaký synchronizace. Jde většinou o místa, ve kterých výpočet přechází do nějaké své další fáze, ve které se mají použít data připravená ve fází předchozí.

Fáze I, II, III paralelního výpočtu pro model „evakuace“ jsou obecným návrhovým vzorem pro modely, ve kterých simulační krok skládá z 1) vnímání okolí a rozhodnutí o nejlepším pohybu, 2) vyjasnění konkurenčních bojů (zde souboj o stejnou cílovou buňku, obecně může jít o jakýkoli sdílený prostředek) a 3) příprava rozhodovacích datových struktur na další simulační krok.

Prostorové multi-agentové modely – sdílení informace o poloze agentů (více agentů na jedné buňce) a konkurenční zápis do sdílených datových struktur. Typickým modelem v NetLogu, který bude využívat prostředky paralelizace pomocí NL2OCL je prostorový multi-agentový model (spacial multi-agent model). V modelech tohoto typu je nutné řešit problém více agentů na jedné buňce. K tomu dochází, pokud model pracuje se spojitými souřadnicemi a v případě, že souřadnice více agentů mohou spadat do oblasti, kterou zaujímá jedna buňky. Informace o poloze agentů musí být v takovém případě uloženy do sdílené datové struktury. Tato struktura je určena jak pro čtení (agenti kontrolují polohu ostatních agentů v nějakém definovaném okolí), tak pro zápis (agenti mohou svoji polohu měnit, se změnou absolutních souřadnic polohy agenta dochází k přechodům mezi buňkami). Výskyt toho problému indikuje popis modelu v protokolu ODD v části koncepty návrhu – individuální vnímání a interakce. Zde je specifikováno, jakým způsobem agent vnímá své okolí, zda získává informaci o okolních agentech a zda může více agentů sdílet prostor jedné buňky. Pokud má být datová struktura pro sdílení informace o poloze využitelná při paralelním zpracování, musí daná datová struktura používat nějaký nezamykatelný způsob konkurenčního čtení a zápisu. Umožňují to wait-free principy a lock-free konstrukty využívající atomické operace, často např. operaci CAS (zkontroluj a změň, check-and-switch). V NL2OCL byla navržena a implementována sdílená datová struktura PA uchovávající informace o obsazenosti jednotlivých buněk agenty. Její lock-free použití je demonstrováno v příkladu paralelizace hejna – viz. kap. 6.2. Další možné řešení je problému předejít, tj. při paralelním zpracování neprovádět žádné konkurenční zápisy do sdílené datový struktury. Ukázka tohoto přístupu je uvedena v popisu paralelizace evakuačního modelu v kap. 6.3.. Tam je způsobem „vyhnutí se problému“ řešen proces rozhodování o vítězi, který obsadí v rámci simulačního kroku buňku, o kterou má zájem více agentů.

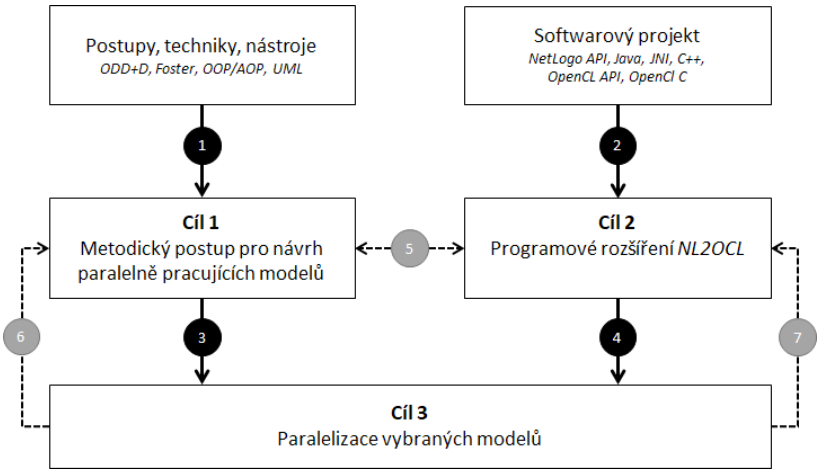
Rozhodování analogické s NetLogo konstrukcí one-of (výběr jednoho náhodného agenta) je možné řešit v OpenCL kernelu pomocí generátoru pseudonáhodných čísel a výběru jednoho prvku při rovnoměrném rozložení pravděpodobnosti. Implementační detaily viz příloha B (OpenCL kernel pro model *EVAKUACE*).

7 Diskuze dosažených výsledků

Výsledky této disertační práce odpovídají třem vytyčeným dílčím cílům. Výsledky jsou různého charakteru, vznikaly postupně a vzájemně se ovlivňovaly. Práce na výstupech probíhala ve dvou hlavních směrech: budování metodického postupu pro návrh paralelně pracujících modelů a vytvoření programového rozšíření NL2OCL. Oba tyto směry se v konečné fázi spojily při paralelizaci vybraných modelů. Obr. 42 ukazuje, jak práce na disertačním výzkumu probíhaly.

Vstupem pro vznik metodického postupu byly vybrané ověřené postupy, techniky a nástroje (obr. 42-1), práce na programovém rozšíření NL2OCL byly odděleným softwarovým projektem zahrnujícím použití celé škály technologií (obr. 42-2). Navržený metodický postup pro paralelizaci multi-agentových modelů a jeho technická realizace byly ověřovány na různých modelech (obr. 42-3), z nichž byly v této práci z prostorových důvodů představeny tři (v kapitolách 6.1, 6.2 a 6.3).

Programové rozšíření NL2OCL se stalo nástrojem, pomocí kterého bylo paralelizace dosaženo prostřednictvím funkcí platformy OpenCL (obr. 42-4). Kromě těchto přímých vazeb mezi jednotlivými cíli se silně uplatnily vazby, kterými se jednotlivé výstupy ovlivňovaly navzájem. Mezi metodickým postupem a implementací programového rozšíření NL2OCL (obr. 42-5) probíhaly iterace, kdy z potřeb metodického postupu vyplývaly požadavky na funkčnost NL2OCL a v obráceném směru, technické možnosti platformy OpenCL a programového rozšíření NL2OCL vytyčovaly hranice, ve kterých se mohl metodický postup rozvíjet, aby byl technicky realizovatelný. Na základě praktických zkušeností z paralelizace vybraných modelů byl upravován jak metodický postup, tak funkcionalita nástroje N2OCL (obr. 42-6, 7).



Obrázek 42- proces vzniku výstupů a vzájemné vazby, zdroj: Autor

První dílčí disertační cíl – byl vytvořen metodický postup pro návrh multi-agentových modelů s paralelními výpočty, který kombinuje různé osvědčené přístupy, techniky a nástroje do jednoho celku. Tento metodický postup dává jednotlivým dílčím aspektům paralelizace multi-agentových

modelů jasnou strukturu. Dostupné práce zabývající se využitím paralelních výpočtů v multi-agentových modelech se věnují technické implementaci jednotlivých paralelně pracujících modelů či některým dílčím aspektům návrhu a vývoje nebo jsou naopak velmi obecné a nedají se použít jako konkrétní návod, jak paralelně pracující model od začátku do konce navrhnout a implementovat (Mittal, 2015; Hermellin & Michel, 2016).

Vytvořený metodický postup se snaží provést tvůrce paralelně pracujících modelů celým procesem návrhu, od specifikace modelu, přes jednotlivé fáze návrhu modelu jako paralelní aplikace až po ověření správnosti fungování modelu, vyladění jeho výkonu a provedení simulací a experimentů nad tímto modelem. Při formulování jednotlivých částí metodického postupu se ukázala jako vhodná forma popisu kroků trojice: otázky → možné odpovědi → promítnutí odpovědí do návrhu modelu. Kromě vytipovaných osvědčených postupů a technik byly do metodiky promítnuty také reálné zkušenosti z paralelizace vybraných modelů za použití NL2OCL.

Druhý dílčí disertační cíl – bylo vytvořeno programové rozšíření NL2OCL umožňující využití paralelních výpočtů prostřednictvím platformy OpenCL. V toto chvíli je NL2OCL uceleným, robustním nástrojem připraveným k okamžitému použití. Za pomoci NL2OCL je možné vyvíjet multi-agentové modely v systému NetLogo, a přitom využívat sílu paralelního zpracování výpočtů.

Pro srovnání v práci (Laville at al., 2014), která se věnuje podobnému tématu, tedy využití prostředků paralelizace modelů prostřednictvím výpočtů v heterogenních výpočetních systémech, autoři představují vytvořenou knihovnu pro multi-agentové systémy MCMAS (Many Core MAS). Jedná se o generickou sadu funkcí pro využití mnoho-jádrových architektur za pomoci definovaných datových struktur a implementace konkrétních kernelů. Implementace MCMAS je provedena jako sada Java tříd. Použití je tedy vázáno na zdrojový program modelu v jazyce Java. MCMAS poskytuje konkrétní implementaci některých vybraných algoritmů jako je difuze, hledání cesty či populační dynamika. Podobnost MCMAS s NL2OCL je ve volbě platformy OpenCL pro její otevřenost a možnost práce v heterogenním výpočetním prostředí (systémy obsahující různé typy výpočetních jednotek, jak CPU, tak GPU). Oproti MCMAS umožňuje NL2OCL větší kontrolu nad tím, jakým způsobem jsou prostředky paralelizace využity, což vede k širším možnostem použití. MCMAS umožňuje rozvoj nové funkčnosti nad rámec již připravených algoritmů cestou vývoje dodatečných Java pluginů využívajících rozhraní JOCL pro zapouzdření funkcionality OpenCL. Programové rozšíření NL2OCL je v tomto směru flexibilnější a poskytuje přímou cestu od agendového modelu k implementaci paralelního zpracování prostřednictvím OpenCL kernelů. Autoři modelů mohou při použití NL2OCL okamžitě používat nově připravené OpenCL kernely a libovolně kernely upravovat. Tato vlastnost NL2OCL je užitečná zejména pro postupné ladění kernelů a umožňuje flexibilně upravovat způsob paralelního výpočtu. Autor

modelu se zabývá jen vlastním modelem a OpenCL kernelem, nemusí se starat o Java mezivrstvu zapouzdřující platformu OpenCL.

Třetí dílčí disertační cíl – bylo demonstrováno použití navrženého metodického postupu a programového rozšíření NL2OCL při implementaci paralelních verzí vybraných modelů. Došlo k ověření, že metodický postup a NL2OCL programové rozšíření jsou v praxi použitelné. Byly implementovány různé typy modelů – model se spojitě pohybujícími se agenty a model na bázi buněčného automatu. Oba modely zahrnovaly implementaci chování agentů, skládajícího se z několika vzorců vnímání, rozhodování a pohybu. Při paralelizaci modelů byly použity jednotlivé kroky metodického postupu. Během doménové a funkční dekompozice došlo k identifikaci těch částí modelu, které se mohou paralelizovat. Byly analyzovány požadavky na komunikaci a navrženy datové struktury pro realizaci této komunikace. Na evakuačním modelu bylo demonstrováno použití neblokující datové struktury s konkurenčním přístupem pro čtení a zápis, pro sdílení informace o poloze agentů. Pro modely byly implementovány kernely pro paralelní zpracování výpočtů. Při implementaci těchto kernelů byly vytypovány části, které se dají snadno znovu použít jako stavební bloky pro realizaci některých typických aktivit agentů.

Provedené experimenty pro paralelizované modely se soustředily na zmapování dosažených zrychlení simulace při použití NL2OCL pro měnící se parametry modelů a při zvětšujícím se rozsahu populace agentů. Jak v modelu *HEJNA*, tak v komplexnějším modelu *EVAKUACE* se projevila dobrá škálovatelnost paralelního řešení (zvýšení rychlosti rostlo se zvyšujícím se rozsahem úlohy).

V modelu *HEJNA* bylo pro populaci 10 000 až 100 000 agentů a pro různé vizuální poloměry r (vzdálenost, na kterou agent dohlédne) dosaženo při použití NL2OCL zrychlení vrozsahu 21–40 (pro r = 3), 74–105 (r = 8) a 130–164 (r = 15).

Pro model *EVAKUACE* byla výsledná zrychlení při použití NL2OCL pro populaci 10 000 až 100 000 agentů 29–90 (pro r = 1), 49–161 (r = 5) a 59–254 (r = 15).

Tyto výsledky umožňují potvrzení hypotézy, že **simulace velkého rozsahu se dají provádět i v prostředí standardních PC za pomoci výpočtů na výkonných grafických kartách**.

Lze srovnávat dosažená zrychlení pomocí NL2OCL se zrychleními, kterých dosahovali jiní autoři, kteří se zabývali paralelizací stejných modelů. Např. (Hermellin & Michel, 2016) popisují zrychlení simulace Reynoldsova modelu hejna pomocí výpočtů na grafické kartě. Ve svém článku uvádějí srovnání dosažených rychlostí simulace pro různé implementace tohoto modelu ve vybraných multi-agentových systémech (včetně NetLoga) oproti rychlosti jejich modelu realizovaném pomocí metody GPU delegace chování agentů na prostředí. Srovnání je provedeno na modelu o velikosti 512 x 512 buněk s populací

4 000 agentů. Z implementací používajících GPU uvádějí autoři platformu Flame GPU. Uvedené zrychlení Flame GPU proti samotnému NetLogu je 2,6krát (simulační krok v NetLogu = 214ms, ve Flame GPU = 82ms). Autoři identifikovali jako část modelu vhodnou pro paralelní zpracování pouze část zodpovědnou za realizaci pohybového vzorce soudržnosti. Autoři využívají platformu CUDA (proprietární řešení pro grafické karty NVIDIA) prostřednictvím knihovny JCuda zprostředkující CUDA funkcionalitu Java aplikacím. Zrychlení, kterého autoři dosáhli bylo 25 % oproti řešení, které nepoužívá GPU výpočty. Zrychlení paralelizovaného modelu hejna pomocí NL2OCL popsaného v kap. 6.1 v prostředí 500 x 500 buněk a pro 10 000 agentů bylo oproti čisté NetLogo implementaci 21násobné. Pro větší populaci agentů dosahuje NL2OCL zrychlení ještě vetší. Tento rozdíl je dán způsobem paralelizace. Model v kap. 6.1 paralelizuje všechny tři složky chování agentů hejna, navíc je díky použitým datovým strukturám optimalizována komunikace. Dosažené výsledky, jednak samotným programovým rozšířením NL2OCL, ale také způsobem, jakým byl model paralelizován, jsou v tomto případě velmi slibné.

Paralelní výpočty provedené v rámci této disertační práce během všech experimentů byly realizovány na grafické kartě AMD Radeon R9 290X (specifikace viz příloha C). Volba této karty byla dána vhodností jejich parametrů a dobrou příležitostí k jejímu získání. Karta pochází z doby těsně po útlumu masivní „těžby“ krypto-měn (zejména Bitcoin, Lightcoin), ke kterému došlo v roce 2015. Po nástupu speciálních zakázkových ASIC čipů (Application Specific Integrated Circuit) se tato činnost stala z ekonomického hlediska neefektivní. Cena za energii nutnou pro běh těžebního systému byla vyšší než cena vytěžené krypto-měny. Mnoho velice výkonných grafických karet pak bylo k dispozici za zlomek jejich pořizovací ceny. I když tyto karty nepocházejí ze segmentu GPU pro speciální aplikace, jedná se o výpočetně velmi silná zařízení se všemi předpoklady k realizování paralelních výpočtů.

V současné době (rok 2017) se situace s grafickými kartami opakuje. Nové druhy kryptoměn, které jsou vhodné pro těžbu na grafických kartách (např. Monero, ZCash a Ethereum) rapidně zvyšují poptávku po kartách nové generace (např. NVIDIA GeForce GTX 1070/1080/1080Ti a AMD Radeon RX 570/580 či novější Vega RX 64/56). V blízké budoucnosti (1-2 roky) dojde zákonitě, kvůli principům zvyšující se složitosti algoritmů kryptoměn, k prolomení hranice efektivity použití, kdy náklady na spotřebu opět převýší výnosy z těžby. Potom se na trhu objeví velké množství silných grafických karet za velmi příznivé pořizovací ceny. Využití těchto karet pro vědecké výpočty bude vhodnou alternativou k nákupu výpočetního času prostřednictvím akcelerovaných výpočetních cloudových služeb (např. Amazon ECP, Google GCP). Vybudování heterogenního výpočetního systému pro běh rozsáhlých multi-agentových simulací by mohlo být přínosným výzkumným projektem. Výstupy této disertační práce (metodický postup a programové rozšíření NL2OCL) mohou takovýmto projektům dobře posloužit.

8 Závěr

V této práci byly prezentovány výstupy disertačního výzkumu zaměřeného na urychlení multi-agentových simulací velkého rozsahu prostřednictvím heterogenních výpočetních systémů.

Ve druhé kapitole byly stanoveny cíle a metodický postup, jak budou tyto cíle naplněny. Ve třetí kapitole byl analyzován současný stav odvětví vztahujících se k možnostem paralelního zpracování výpočtů v heterogenních výpočetních systémech. Poté následovaly tři kapitoly odpovídající třem dílčím cílům disertace. Ve čtvrté kapitole šlo o představení **vytvořeného metodického postupu pro návrh a implementaci multi-agentových modelů s paralelně zpracovanými výpočty**. Pátá kapitola popisovala **vytvořené programové rozšíření NL2OCL** umožňující modelům v systému NetLogo využít paralelní výpočty na grafických kartách. Nakonec byla v šesté kapitole **ověřena použitelnost na vybraných multi-agentových modelech**. Pro tři modely bylo navrženo, jakým způsobem mohou využít paralelní zpracování výpočtů pro urychlení simulací. Dva z těchto modelů byly implementovány jako paralelně pracující verze. Byly navrženy a provedeny experimenty ověřující dosažené zrychlení při použití NL2OCL.

Výsledky jsou detailně diskutovány v kapitole 7. **Při použití NL2OCL se zvýšení rychlosti simulace pohybovalo od 20ti násobného** (v modelu *HEJNA* pro populaci 10 000 agentů) **až po 250ti násobné** (v modelu *EVAKUACE* pro 100 000 agentů a vizuální rádius o velikosti 10 buněk) v porovnání s rychlostí, kterou pro stejné parametry modelu disponovalo samotné NetLogo. Na základě těchto výsledků **bylo možné potvrdit hypotézu, že se simulace velkého rozsahu dají provádět i v prostředí standardních PC za pomoci výpočtů na výkonných grafických kartách.**

Výstupy této disertační práce otevírají prostor k dalšímu výzkumu. Jedná se zejména o možnost vytvářet k existujícím modelům jejich paralelně pracující verze a tím umožnit podstatně větší a rychlejší simulace nad těmito modely. Navazující výzkum by se také mohl týkat rozšíření NL2OCL o funkcionalitu využívající principy prostorové indexace (*spatial indexing*), které umožňují velmi rychlý přístup k informacím o poloze agentů v rozsáhlém prostřední modelu. NL2OCL disponující takovou funkcionalitou by bylo vhodným nástrojem pro modely pracující s rozsáhlými GIS daty nebo pro jakékoli jiné prostorové multi-agentové modely s velmi rozsáhlým prostředím. Také se nabízí možnost na základě představeného modelu *EVAKUACE* (a připravených OpenCL kernelů) vybudovat modely s komplexnějšími evakuačními scénáři a provádět nad nimi rozsáhlé simulace. V neposlední řadě bude zajímavé využít možnosti NL2OCL v jiných multi-agentových systémech postavených na platformě Java.

9 Seznam použité literatury

Aljabri, M., Belikov, E., Deligiannis, P., Loidl, H., & Totoo, P. (2013). A Survey of High-Level Parallel Programming Models. Technical report. School of Mathematical and Computer Sciences. Heriot/Watt University, Edinburgh.

Almasi, G. S., & Gottlieb, A. (1989). Highly Parallel Computing. New York, NY, USA: Benjamin-Cummings publishers.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS 67.

Angelotti, E., Scalabrin, E., & Avila, B. (2001). PANDORA: a multi-agent system using paraconsistent logic. Proceedings Fourth International Conference on Computational Intelligence and Multimedia Applications. ICCIMA 2001. 352-356.

Asanovic, K., Bodik, R., Catanzaro, B.R., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf. J., Williams, & S.W, Yelick, K.A. (2006). The Lanscape of Parallel Computing Research: A view from Berkeley. Research Report. University of California at Berkeley.

Berezin, P. (ed.) (2013). Human Intelligence And Economic Growth From 50,000 B.C. To The Singularity. BCA research - Special Report.

Brit, H., & Moran, S. (1994). Wait-freedom vs. bounded wait-freedom in public data structures (extended abstract). In Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing ACM. 52-60.

Burstedde, C., Klauck, K., Schadschneider, A. & Zittartz, J. (2001). Simulation of pedestrian dynamics using a two-dimensional cellular automaton. Physica A: Statistical Mechanics and its Applications.295(3-4).507-525.

Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M. & Tsigas, P. (2017) Lock-Free Concurrent Data Structures, in Programming multi-core and many-core computing systems (eds S. Pllana and F. Xhafa). Hoboken, NJ: Wiley & Sons

Collier, N. and North, M. (2012) Repast HPC: A Platform for Large-Scale Agent-Based Modeling, in Large-Scale Computing (eds W. Dubitzky, K. Kurowski and B. Schott). 81-109. Hoboken, NJ, USA: John Wiley & Sons, Inc.

Cordasco, G., Chiara, R. D., Mancuso, A., Mazzeo, D., Scarano, V., & Spagnuolo, C. (2012). A Framework for Distributing Agent-Based Simulations. Euro-Par 2011: Parallel Processing Workshops Lecture Notes in Computer Science. 460-470.

Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C. (2012). Exploitation of hpc in the flame agent-based simulation framework. In: Proceedings of the 2012 IEEE 14th Int. Conf. on HPC and Communication. 538–545.

Coffman, E. G., Elphick, M., & Shoshani, A. (1971). System Deadlocks. ACM Computing Surveys (CSUR). 3(2). 67-78.

CPU-Z (2016) OC World Records - Highest Overclocking of All Times. Online: https://valid.x86.fr/records.html

Davidsson, P. (2002). Agent Based Social Simulation: A Computer Science View. Journal of Artificial Societies and Social Simulation. 5 (1).

Deissenberg, C.H., Hoog, V.D., & Dawid, H. (2008). EURACE: Massively Parallel Agent-Based Model of the European Economy. Special Issue on New Approaches in Dynamic Optimization to Assessment of Economic and Environmental Systems. 204(2). 541-552.

Duives, D.C., Daamen, W., & Hoogendoorn, S.P. (2013). State-of-the-art crowd motion simulation models. Transportation Research Part C: Emerging Technologies. 37(Supplement C). 193-209.

Durkin, T. (1998). The Vx_files: What the media couldn't tell you about Mars PathFinder. Robot Science and Technology. 1. 1-3.

Fang, J., Varbanescu, A.L., & Sips, H.J. (2011). A Comprehensive Performance Comparison of CUDA and OpenCL. 2011 International Conference on Parallel Processing. 216-225.

Flynn, M.J. (1966). Very High-Speed Computing Systems. Proceedings of the IEEE. 54(12). 1901-1909.

Foster, I. (1995). Designing and building parallel programs: concepts and tools for parallel software engineering. Reading, MA: Addison-Wesley.

Fuller, S. H., & Millett, L. I. (2011). The future of computing performance: game over or next level?. Washington, D.C.: National Academies Press.

Gilbert, N., & Troitzsch, K. G. (2011). Simulation for the social scientist. Maidenhead: Open Univ. Press.

Grimm, V., Berger, U., DeAngelis, D.L., Polhill, J.G., Giske, J., & Railsback, S.F. (2010). The ODD protocol: A review and first update. Ecological Modelling. 221(23). 2760-2768.

Grimm, V., & Railsback, S.F. (2005). Individual-based Modeling and Ecology. Princeton University Press.

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S., Huse, G., Huth, A., Jepsen, J., Jørgensen, C., Mooij, W., Müller, B., Pe'er, G., Piou, C., Railsback, S., Robbins, A., Robbins, M., Rossmanith, E., Rüger, N., Strand, E., Souissi, S., Stillman, R., Vabø, R., Visser, U. and DeAngelis, D. (2006). A standard protocol for describing individual-based and agent-based models. Ecological Modelling, 198(1-2). 115-126.

Grimm, V., Berger, U., DeAngelis, D., Polhill, J., Giske, J. and Railsback, S. (2010). The ODD protocol: A review and first update. Ecological Modelling. 221(23). 2760-2768.

Gustafson, J.L. (1988). Reevaluating Amdahl's Law. Communications of the ACM. 31(5). 532-533.

Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. San Francisco, CA: Morgan Kaufmann.

Heywood, P., Richmond, P., & Maddock, S. (2015). Road Network Simulation Using FLAME GPU. Euro-Par 2015: Parallel Processing Workshops Lecture Notes in Computer Science. 430-441.

Horling, B., & Lesser, V.A. (2004). Survey of Multi-Agent Organizational Paradigms. The Knowledge Engineering Review. 19(4). 281-316.

Helbing, D. & Johansson, A. (2011). Pedestrian, Crowd and Evacuation Dynamics. Encyclopedia of Complexity and Systems Science. 13. 697–716.

Herlihy, M., & Shavit, N. (2008). The Art of Multiprocessor Programming. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Hermellin, E., & Michel, F. (2016). GPU Environmental Delegation of Agent Perceptions: Application to Reynolds's Boids. Multi-Agent Based Simulation XVI Lecture Notes in Computer Science. 71-86.

Hermellin, E., & Michel, F. (2016). Overview of Case Studies on Adapting MABS Models to GPU Programming. Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection Communications in Computer and Information Science. 125-136.

International Technology Roadmap for Semiconductors (ITRS2009). Retrieved September 24, 2017, from http://www.itrs.net/links/2009ITRS/ Home2009.htm.

Kaeli, D.R., Mistry, P., Schaa, D., & Zhang, D.P. (2015) Heterogeneous Computing with OpenCL 2.0. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Karp, A.H., & Flatt, H.P. (1990). Measuring Parallel Processor Performance. Communication of the ACM. 33(5). 539-543.

Kirchner, A., Klüpfel, H., Nishinari, K., Schadschneider, A. & Schreckenberg, M. (2004). Discretization effects and the influence of walking speed in cellular automata models for pedestrian dynamics. Journal of Statistical Mechanics: Theory and Experiment. 2004(10).

Koomey, J.G., Berard, S., Sanchez, M., & Wong, H. (2009). Assessing trends in the electrical efficiency of computation over time. Annals of the History of Computing.

Kravari, K., & Bassiliades, N. (2015). A Survey of Agent Platforms. Journal of Artificial Societies and Social Simulation. 18(1). Online: http://jasss.soc.surrey.ac.uk/18/1/11.html.

Laborde, P., Feldman, S., & Dechev, D. (2017). A Wait-Free Hash Map. International Journal of Parallel Programming. 45. 421-448.

Laville, G., Mazouzi, K., Lang, C., Marilleau, N., & Philippe, L. (2012). Using GPU for Multi-agent Multi-scale Simulations. Distributed Computing and Artificial Intelligence: 9th International Conference. 197-204.

Laville, G., Mazouzi, K., Lang, C., Marilleau, N., Herrmann, B., & Philippe, L. (2014). MCMAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation. Euro-Par 2013: Parallel Processing Workshops Lecture Notes in Computer Science. 544-554.

Lee, J., Keo. M., Park, I., Chung, J. (2010). The Rotated Hexagonal Lattice Model for Pedestrian Flows. Proceedings of the Eastern Asia Society for Transportation Studies. 8. 1357-1367.

Leng, B., Wang, J. & Xiong, Z. (2015). Pedestrian simulations in hexagonal cell local field model. Physica A: Statistical Mechanics and its Applications. 438. 532-543.

Lis, M., Shim, K. S., Cho, M. H., & Devadas, S. (2011). Memory coherence in the age of multicores. IEEE 29th International Conference on Computer Design (ICCD). 1-8.

Liu, Z., Calciu, I., Herlihy, M., & Mutlu, O. (2017). Concurrent Data Structures for Near-Memory Computing. Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures. 235-245.

Lu, L., Chan, C., Wang, J. & Wang, W. (2017). A study of pedestrian group behaviors in crowd evacuation based on an extended floor field cellular automaton model. Transportation Research Part C: Emerging Technologies. 81. 317-329.

Lu, L., Ren, G., Wang, W. & Wang, Y. (2014). Modeling walking behavior of pedestrian groups with floor field cellular automaton approach. Chinese Physics B. 23(8).

Marowka, A. (2016). Energy-Aware Modeling of Scaled Heterogeneous Systems. International Journal of Parallel Programming. 45(5). 1026-1045.

Martin, M. M., Hill, M. D., & Sorin, D. J. (2012). Why on-chip cache coherence is here to stay. Communications of the ACM. 55(7). 78-89.

Michael, M. (2015). Non-Blocking Synchronization and Memory Management. Applicative 2015 on - Applicative 2015.

MWC64X - Uniform random number generator for OpenCL. (2015). url: http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html (visited on 07/07/2017).

Macal, C.M., North, M.J. (2005). Tutorial on agent-based modeling and simulation. Proceedings of the 37th Winter Simulation Conference. 2-15.

Macal, C.M., North, M.J. (2006). Tutorial on agent-based modeling and simulation part 2: how to model with agents. Proceedings of the 38th Winter Simulation Conference. 73-83.

Macal, C.M., North, M.J. (2010). Tutorial on agent-based modelling and simulation. Journal of Simulation. 4. 151–162.

Merrit, R. (2016). Moore's Law Goes Post-CMOS. Retrieved September 24, 2017, from http://www.eetimes.com/document.asp?doc_id=1328835.

Mittal, S. & Vetter, J. (2015). A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Computing Surveys. 47(4). 1-35.

Moor, G.E. (2006). Cramming more components onto integrated circuits. Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. IEEE Solid-State Circuits Society Newsletter. 11(5). 33-35.

Müller, B., Bohn, F., Dreßler, G., Groeneveld, J., Klassert, C., Martin, R., Schlüter, M., Schulze, J., Weise, H. and Schwarz, N. (2013). Describing human decisions in agent-based models – ODD + D, an extension of the ODD protocol. Environmental Modelling & Software. 48. 37-48.

Nikolakopoulos, Y., Gidenstam, A., Papatriantafilou, M., & Tsigas, P. (2015). Of Concurrent Data Structures and Iterations. In: Zaroliagis C., Pantziou G., Kontogiannis S. (eds) Algorithms, Probability, Networks, and Games. Lecture Notes in Computer Science. 9295. 358-369

Pavlov, R., & Müller, J. P. (2013). Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors. IFIP Advances in Information and Communication Technology Technological Innovation for the Internet of Things. 115-122.

Pei, S., Zhang, J., Xiong, N., Kim, M., & Gaudiot, J. (2015). Performance-energy efficiency model of heterogeneous parallel multicore system. Sixth International Green and Sustainable Computing Conference (IGSC). 1-6.

Pirkelbauer, P., Milewicz, R., & Felipe, G.J. (2016). A Portable Lock-Free Bounded Queue. Algorithms and Architectures for Parallel Processing: 16th International Conference, ICA3PP. 55-73.

Pllana, S., & Fatos, X. (eds) (2017). Programming Multicore and Many-Core Computing Systems. Wiley Series On Parallel and Distributed Computing. Hoboken, New Jersey: Wiley & Sons

Polhill, J.G., Parker, D., Brown, D., & Grimm, V. (2008). Using the ODD Protocol for Describing Three Agent-Based Social Simulation Models of Land-Use Change. Journal of Artificial Societies and Social Simulation.11(2),3.Online: http://jasss.soc.surrey.ac.uk/11/2/3.html

Ramapantulu, L., Tudor, B. M., Loghin, D., Vu, T., & Teo, Y. M. (2014). Modeling the Energy Efficiency of Heterogeneous Clusters. 43rd International Conference on Parallel Processing. 321-330.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C. & Thiele, J. (2017). Improving Execution Speed of Models Implemented in NetLogo. Journal of Artificial Societies and Social Simulation. 20(1), 3. Online: http://jasss.soc.surrey.ac.uk/20/1/3.html.

Railsback, S., & Grimm, V. (2011). Agent-Based and Individual-Based Modeling: A Practical Introduction. Princeton University Press.

Railsback, S.F., Lytinen, S.L., & Jackson, S.K. (2006). Agent-based Simulation Platforms: Review and Development Recommendations. Simulation. 82(9). 609-623.

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. Proceedings of the 14th annual conference on Computer graphics and interactive techniques. 25-34.

Ricci, A., Croatti, A., Brunetti, P., & Viroli, M. (2015). Programming Mirror Worlds: An Agent-Oriented Programming Perspective. Engineering Multi-Agent Systems Lecture Notes in Computer Science. 191-211.

Rodríguez, V.M. & Álvarez, G.D. (2015). Parallelism in bioinformatics: A view from different parallelism-based technologies. Parallel Computing. 42.1-3.

Rousset, A., Herrmann, B., Lang, Ch., Philippe, L. (2014). A Survey on Parallel and Distributed Multi-Agent Systems. In: Lopes, L. et al. (eds) Euro-Par 2014 Workshops, Part I, LNCS. 8805. 371–382.

Sahni, S., & Thanvantri, V. (1995). Parallel Computing: Performance Metrics and Models. Research Report. Computer Science Department. University of Florida.

Sha, L., Rajkumar, R. & Lehoczky, J. (1990). Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers. 39(9). 1175-1185.

Shoham, Y. (1993). Agent-oriented programming. Artificial Intelligence. 60(1). 51-92.

Siegfried, R. (2014). Parallel and distributed multi-agent simulation. Modeling and Simulation of Complex Systems. 49-60.

Souza, A.M., Pereira, F.D., & Ordonez, E.D.M. (2013). Exploiting Heterogenous Systems: Keccak on OpenCL. International Conference on Parallel and Distributed Processing Techniques and Applications.

Steele, G.L. (2009). Organizing Functional Code for Parallel Execution. Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. 1-2.

Techpowerup AMD Radeon R9 290X specification. Retrieved September 24, 2017, from https://www.techpowerup.com/gpudb/2460/radeon-r9-290x

The OpenCL Specification - Khronos Group. Retrieved September 24, 2017, from https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdfAccessed 24 Sept. 2017.

Timnat, S. & Petrank, E. (2014). A practical wait-free simulation for lock-free data structures. ACM SIGPLAN Notices. 49(8). 357-368.

Ulhrmacher, A.M., Weyns, D. (eds.) (2009). Multi-Agent Systems: Simulations and Applications. CRC Press.

Wilensky, U. (1998). NetLogo Flocking model. http://ccl.northwestern.edu/netlogo/models/Flocking. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Wirth, N. (1978). Algorithms + Data Structures = Programs. Prentice Hall PTR

Yang, X., Wang, B. & Qin, Z. (2015). Floor Field Model Based on Cellular Automata for Simulating Indoor Pedestrian Evacuation. Mathematical Problems in Engineering. 2015. 1-10.

Zhang, D. & Dechev, D. (2016). A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. IEEE Transactions on Parallel and Distributed Systems. 27(3). 613-626.

10 Vlastní publikace vztahujících se k disertačnímu tématu

Procházka, J., & Štekerová, K. (2017). OpenCL for Large-Scale Agent-Based Simulations. Computational Collective Intelligence Lecture Notes in Computer Science. 351-360. Cham: Springer International Publishing.

Olševičová, K., Procházka, J., & Danielisová, A. (2015). Reconstruction of Prehistoric Settlement Network using Agent-based Model in NetLogo. *Proceedings Paper. 13th International Conference on Practical Applications of Agents, Multi-Agent Systems, and Sustainability (PAAMS)*. 165-175. Cham: Springer International Publishing.

Procházka, J., & Olševičová, K. (2015). Monitoring Lane Formation of Pedestrians: Emergence and Entropy. Proceedings Paper. 7th Asian Conference on Intelligent Information and Database Systems (ACIIDS). 221-228. Cham: Springer International Publishing.

Procházka, J., Cimler, R., & Štekerová, K. (2015). Pedestrian Modelling in NetLogo. Emergent Trends in Robotics and Intelligent Systems. 303-312. Cham: Springer International Publishing.

Štekerová, K., & Procházka, J. (2015). Exploration of Prehistoric Settlement Networks using the Minimum Spanning Tree. Proceedings Paper. 26th International-Business-Information-Management-Association Conference.

Procházka, J., Tesařová, B., & Štekerová, K. (2015). Pedestrian Crowd Dynamics: from Agents to Self-Organization. Proceedings Paper; 26th International-Business-Information-Management-Association Conference.

Procházka, J., & Olševičová. K. (2014). Agent-based Model of Spatial Dispersion of Celtic Settlements. Advances in informatics, information management and administration. Technická univerzita v Liberci.

Zelenka, J., Olševičová, K., Cimler, R., Pásková, M., Procházka, J. (2014). Aplikace umělé inteligence a kognitivní vědy v udržitelnosti cestovního ruchu. Hradec Králové: Gaudeamus.

Procházka, J., & Olševičová, K. (2014). Data-Driven Pedestrian Model: From OpenCV to NetLogo. Proceedings Paper. 6th International Conference on Computational Collective Intelligence (ICCCI). 322-331. Cham: Springer International Publishing.

Cimler, R., Čech, P., Kysela, J., Olševičová, K., Procházka, J., Štýrský, J., & Vítek, O. (2014). *Modul kognitivních, informatických a multioborových přístupů k udržitelnosti cestovního ruchu*. Hradec Králové: Gaudeamus.

Procházka, J. (2012). Tabulová architektura jako metoda spolupráce v multiagentových systémech. *Sborník příspěvků z letní školy „Mezioborové přístupy informatiky a kognitivní vědy“*. Online: http://fim.uhk.cz/inkov/doc/LS_2012_sbornik_final.pdf.

Příloha A – ODD+D protokol

V tabule 12 je uvedeno plné znění položek protokolu ODD+D a otázek spojených s jednotlivými položkami. Tučně jsou vyznačeny rozšíření protokolu (+D) která jsou relevantní pro individuální rozhodování agentů.

Tabulka 12 – kompletní protokol ODD + D, zpracováno podle (Grimm et al., 2010; Müller at al. 2013)

Element protokolu		Naváděcí otázka	#
Popis **Overview**	Účel	Jaký je účel modelu?	1.1.1
		Pro koho je model určen?	1.1.2
	Entity Stavové veličiny Rozměry	Jaké entity v modelu vystupují?	1.2.1
		Jaké atributy (stavové veličiny) charakterizují jednotlivé entity?	1.2.2
		Jaké vnější faktory ovlivňují chování modelu?	1.2.3
		Jak je do modelu zahrnut prostor?	1.2.4
		Jaké jsou časové a prostorové rozměry a měřítka?	1.2.5
	Procesy a časování	Co dělají jednotlivé entity, kdy, v jakém pořadí?	1.3.1
Koncepty návrhu **Design Concepts**	Teoretické a empirické předpoklady	Z jakých obecných principů, teorií a hypotéz model vychází?	2.1.1
		Na základě jakých předpokladů se agenti rozhodují?	2.1.2
		Jaký je použitý konkrétní rozhodovací model?	2.1.3
		Pokud je rozhodovací model řízený daty, odkud pocházejí tato data?	2.1.4
		Na jaké úrovni agregace byla tato data posbírána?	2.1.5
	Individuální rozhodování	**Jaké jsou subjekty a objekty rozhodování? Na jaké úrovni je rozhodování agregováno? Je rozhodování víceúrovňové?**	2.2.1
		Jaké je uvažování, na základě kterého se agent rozhoduje? Jsou cíle agentů, o než usilují jednoznačné nebo existují další, dílčí kritéria a cíle?	2.2.2
		Jak přesně agenti rozhodnutí provádějí?	2.2.3
		Přizpůsobují agenti svoje rozhodování vnitřním nebo vnějším faktorům? Pokus ano tak jak?	2.2.4
		Hrají při rozhodování nějakou roli sociální pravidla a kulturní hodnoty?	2.2.5
		Mají na rozhodování vliv nějaké prostorové aspekty?	2.2.6
		Mají na rozhodování vliv nějaké dočasně trvající aspekty?	2.2.7
		Jak je do rozhodování zahrnuta nahodilost?	2.2.8
	Učení	Je v modelu uplatněno individuální učení? Jak agenti v čase mění svoje rozhodování na základě získaných znalostí?	2.3.1
		Uplatňuje se v modelu kolektivní učení?	2.3.2
	Individuální vnímání	Jaké **vnitřní a vnější** stavové hodnoty agenti vnímají a jak tyto ovlivňují jejich rozhodování? **Modeluje proces vnímání nějakou chybovost?**	2.4.1
		Jaké stavové hodnoty a jakých ostatních agentů může agent vnímat? **Je toto vnímání zatíženo chybou?**	2.4.2
		Jaký je prostorový dosah vnímání?	2.4.3
		Jsou mechanizmy, jakými se agenti dostávají k informacím z vnímání explicitní nebo se předpokládá, že tyto informace znají?	2.4.4

		Jsou náklady na vnímání a sběr informací zahrnuty explicitně do modelu?	2.4.5
	Individuální predikce	**Jaká data agenti používají k predikci budoucích podmínek?**	2.5.1
		Jaké interní modely agent používá pro odhad budoucích podmínek nebo budoucích dopadů svých rozhodnutí?	2.5.2
		Mohou se agenti ve svých předpovědích mýlit, jak je to implementováno?	2.5.3
	Interakce	Jsou interakce mezi agenty přímé nebo nepřímé?	2.6.1
		Na čem interakce závisejí?	2.6.2
		Pokud interakce zahrnují komunikaci, jak je taková komunikace prováděna?	2.6.3
		Pokud existuje koordinační síť, jak ovlivňuje chování agentů? Je struktura této sítě předem daná nebo se tvoří během simulace?	2.6.4
	Skupinové chování	Formují agenti nějaká uskupení, která ovlivňují a jsou ovlivněna jednotlivými agenty? Jsou tato uskupení vynucena návrhem nebo vyvstávají během simulace?	2.7.1
		Jak je skupinové chování reprezentováno?	2.7.2
	Různorodost	**Jsou agenti různorodí? Pokud ano, v jakých stavových hodnotách a v jakých procesech se od sebe odlišují?**	2.8.1
		Jsou agenti různorodí v rozhodovacích procesech? Pokud ano, v jakých rozhodovacích modelech a rozhodovacích objektech?	2.8.2
	Nahodilost	Jaké procesy (včetně inicializace) jsou modelovány jako úplně nebo částečně stochastické?	2.9.1
	Pozorování	Jaká data jsou během simulace sbírána pro další testování, pochopení, analýzu? Jak, kdy?	2.10.1
		Jaké celkové výsledky, výstupy nebo vlastnosti modelu vyvstávají na základě individuálního chování agentů? (Emergence)	2.10.2
Detaily	Detaily implementace	Jak byl model naimplementován?	3.1.1
		Je model dostupný, pokud ano, kde?	3.1.2
	Inicializace	Jaký je počáteční stav prostředí modelu, tzn. v čase t = 0?	3.2.1
		Je počáteční stav pro každý běh simulace stejný nebo se pro každý simulační běh mění?	3.2.2
		Jsou počáteční hodnoty voleny libovolně nebo na základě nějakých dat?	3.2.3
	Vstupní data	Používá model vstupní data z nějakých externích zdrojů (např. souborů nebo jiných modelů)?	3.3.1
	Sub-modely	Jaké jsou detaily sub-modelů reprezentujících procesy popsané v části procesy a časování?	3.4.1
		Jaké jsou parametry sub-modelů, jejich rozměry a referenční hodnoty?	3.4.2
		Jak jsou použité sub-modely navrženy nebo vybrány, jak byly parametrizovány a testovány?	3.4.3

Příloha B – Ukázky zdrojových kódů

Výpis OpenCL kernelu pro model HEJNA:

```
//------------------------------------------------------------------------------------------------------------------
// Angles manipulations
#define DEG2RAD 0.01745329
#define RAD2DEG 57.295779

/**
 * Transformation of standard angles into NetLogo angles
 */
float getNetLogoHeading(float heading) {
  float nlHeading = 90 - heading;
  if (nlHeading < 0) {
    nlHeading = 360 - nlHeading;
  }
  return nlHeading;
}

/**
 * Function for two angles subtractions
 */
float subtractHeadings(float h1, float h2){
  float sh = h1 - h2;
  if (sh > 180) {
    sh -= 360;
  } else if (sh < -180) {
    sh += 360;
  }
  return sh;
}

/**
 * Returns limited turn angle
 */
float turnAtMost(float heading, float turn, float maxTurn) {
  if (fabs(turn) > maxTurn) {
    if (turn > 0) {
      return heading + maxTurn;
    } else {
      return heading - maxTurn;
    }
  } else {
    return heading + turn;
  }
}

/**
 * Returns limited heading after turning towards
 */
float turnTowards(float origHeading, float towardsHeding, float maxTurn) {
  return turnAtMost(origHeading, subtractHeadings(towardsHeding, origHeading), maxTurn);
}

/**
 * Returns limited heading after turning awaz
 */
float turnAway(float origHeading, float awayFromHeding, float maxTurn) {
  return turnAtMost(origHeading, subtractHeadings(origHeading, awayFromHeding), maxTurn);
}

/**
 * Main Kernel function
 */
__kernel void move(

  // Memory buffers
  __global int *PX,                  // Agent's patch - x coordinate
  __global int *PY,                  // Agent's patch - y coordinate
  __global float *X,                 // Agent's absolute x coordinate
  __global float *Y,                 // Agent's absolute y coordinate
  __global float *DX,                // Agent's absolute dx
  __global float *DY,                // Agent's absolute dy
  __global float *H,                 // Agent's heading
  __global int *PA,                  // PA array
  __global int *PAI,                 // Array of positions of agents in PA array

  // environment related constants
  const int c,                       // agents count
  const int pa_slot_len,             // length of PA slot
  const int pw,                      // world width - in patches (int)
  const int ph,                      // world height - in patches (int)
  const float w,                     // world width - absolute
```

```
  const float h,                    // world height - absolute
  const float min_x,                // min x coordinate
  const float max_x,                // max x coordinate
  const float min_y,                // min y coordinate
  const float max_y,                // max y coordinate
  const int min_px,                 // min px coordinate
  const int max_px,                 // max px coordinate
  const int min_py,                 // min py coordinate
  const int max_py,                 // max py coordinate
  const int r,                      // vision
  const int r2,                     // square of vision (not to calculate it)
  const float separte_dist2,        // square of min separation
  const float maxSeparateTurn,      // max separation angle
  const float maxAlignTurn,         // max allign turn angle
  const float maxCohereTurn,        // max scohere turn angle
  const float step

){

// get id of the current thread
int id = get_global_id(0);
// kernel function is processed only for defined number of agents
bool bProcess = (id < c);

// local variables
bool bSeparate = false;
float sumddx = 0;
float sumddy = 0;
int min_ai;
int ai;
float sum_hdx = 0;
float sum_hdy = 0;
float sumx = 0;
float sumy = 0;
float sumdx = 0;
float sumdy = 0;
float tms;
float sumth_dx = 0;
float sumth_dy = 0;
int count_hd = 0;
float subhed;
float turn;
float aa = 0;
float cur_H;
bool bTurnPos;
int nei_c = 0;
float min_h;

// for indexes our of agent count kernel does nothing
if (bProcess)     {
  float min_d2 = separte_dist2;
  int cr, cc;
  int pai_s, pai_e, pai;
  int dpr, dpc;
  bool bVUW = false; bool bVDW = false; // wrap flags - vertical up and down
  bool bHLW = false; bool bHRW = false; // wrap flags - horizontal left and right
  float dx, dy;
  float d2;

  // square area
  for (dpr = -1 * r; dpr <= r; dpr++) {
    for (dpc = -1 * r; dpc <= r; dpc++) {
      // we want only circle radius area
      if ((dpr * dpr + dpc * dpc) < r2){
         cr = PY[id] + dpr;
         cc = PX[id] + dpc;
         // world wraps
         if (cr < min_py) {
           cr += ph;
           bVDW = true;
         } else if (cr > max_py) {
           cr -= ph;
           bVUW = true;
         }
         if (cc < min_px) {
           cc += pw;
           bHLW = true;
         } else if (cc > max_px) {
           cc -=pw;
           bHRW = true;
         }

         // check info about agents on the given patch
         // info is stored in PA
```

```
          pai_s = ((cr - min_py) * pw + cc - min_px) * pa_slot_len;
          pai_e = pai_s + pa_slot_len;

          // take all agents from the PA slot = all agents here
          for (pai = pai_s; pai < pai_e; pai++){
            ai = PA[pai];
            if (ai != -1 && ai != id) { // not empty and not me
            dy = Y[ai] - Y[id];
            if (bVDW) {
              dy -= h;
            } else if (bVUW) {
              dy += h;
            }
            dx = X[ai] - X[id];
            if (bHLW) {
              dx -= w;
            } else if (bHRW) {
              dy += w;
            }
            // square of distance
            d2 = (dx * dx) + (dy * dy);
            if (d2 < r2) {
              // Separate?
              if (d2 < min_d2) {
                min_d2 = d2;
                min_h = H[ai];
                bSeparate = true;
              }
              // Align
              sumdx += DX[ai];
              sumdy += DY[ai];

              // Cohere
              sumddx += dx;
              sumddy += dy;

              nei_c++;
            }
          }
        }
      }
    }
  }

  // ----------------- Prepare the move based on the three movement rules --------
  if (bSeparate) {
    // **** SEPARATE ****
    // turn-away ([heading] of nearest-neighbor) maxSeparateTurn
    H[id] = turnAway(H[id], min_h, maxSeparateTurn);
  } else {
    if (nei_c > 0) {
      // **** ALIGN ****
      if (sumdx != 0 && sumdy != 0) {
        // standard angle
        float avgFlockmateHeading = RAD2DEG * atan2(sumdx, sumdy);
        H[id] = turnTowards(H[id], getNetLogoHeading(avgFlockmateHeading), maxAlignTurn);
      }

      // **** COHERE ****
      if (sumddx != 0 && sumddy != 0) {
        float avgHeadingTowardsFlockmates = RAD2DEG * atan2(sumddx / nei_c, sumddy / nei_c);
        H[id] = turnTowards(H[id], getNetLogoHeading(avgHeadingTowardsFlockmates), maxCohereTurn);
      }
    }
  }
}

// ----------------- Synchronization is needed before actual moves are done !!! -------------------
barrier(CLK_GLOBAL_MEM_FENCE);
// ---------------------------------------------------------------------------------------

// only for agent's threads
if(bProcess) {
  // MOVE
  // heading x and y components
  DX[id] = sin(H[id] * DEG2RAD);
  DY[id] = cos(H[id] * DEG2RAD);

  // do agent's step
  X[id] += DX[id] * step;
  Y[id] += DY[id] * step;

  // check world wrapping
  if (X[id] < min_x) {
    X[id] = pw + X[id];
```

```
      } else if (X[id] > max_x) {
        X[id] = -pw + X[id];
      }
      if (Y[id] < min_y) {
        Y[id] = ph + Y[id];
      } else if (Y[id] > max_y) {
        Y[id] = -ph + Y[id];
      }

      // agent's patch coordinates are its rounded absolut coordinates
      int new_px = round(X[id]);
      int new_py = round(Y[id]);

      // check whether patch has been changed
      if (PX[id] != new_px || PY[id] != new_py) {
      // update agent patch position
      PX[id] = new_px;
      PY[id] = new_py;

      // New PA slot for this agent corresponding to its new patch
      int npi = ((new_py - min_py) * pw + new_px - min_px);
      int pai = npi * pa_slot_len;
      int pai_e = pai + pa_slot_len;

      // find empty space for this agent within PA slot belonging to new patch
      for(int i = pai; i < pai_e; i++) {
        if (atomic_cmpxchg(&PA[i], -1, id) == -1) {
          // release agent's current position
          PA[PAI[id]] = -1;
          // aquire new position
          PAI[id] = i;
          break;
        }
      }
    }
  }
}
//------------------------------------------------------ END OF KERNEL FUNCTION ------------------------------------------------------
```

Výpis OpenCL kernelu pro model EVAKUACE:

```
//------------------------------------------------------------------------------------------------------------
#define IS_IN_RANGE(x, y) (x >= min_px && x <= max_px && y >= min_py && y <= max_py)
#define CONTENT_EMPTY   -1
#define CONTENT_WALL    -2
#define CONTENT_DOORS   -3

uint MWC64X(uint2* rvec) {  //Adapted from http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html
  #define A 4294883355U
  uint x=rvec->x, c=rvec->y; //Unpack the state
  uint res = x ^ c;          //Calculate the result
  uint hi = mul_hi(x,A);     //Step the RNG
  x = x*A + c;
  c = hi + (x<c);
  *rvec = (uint2)(x,c);      //Pack the state back up
  return res;                //Return the next result
  #undef A
}
inline float rand_float(uint2* rvec) {
  return (float)(MWC64X(rvec)) / (float)(0xFFFFFFFF);
}

__constant int dirs[2][6][2] = {
  { {1, 0}, {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}, {0, 1} },
  { {1, 0}, {1, -1}, { 0, -1}, {-1, 0}, { 0, 1}, {1, 1} }
};

__kernel void move(
  // pedestrian
  __global int *PX,      // Agents patch x-coordinates
  __global int *PY,      // Agents patch y-coordinates
  __global int *Dead,    // where to move - pycor
  __global int *RND1,    // where to move - pycor
  __global int *RND2,    // where to move - pycor
  __global int *last_dir,// source numbers for random numbers generator
  __global float *X,     // source numbers for random numbers generator
  __global float *Y,     // source numbers for random numbers generator

  // patch
  __global int *C,       // content of patches id of agent or -1 for empty, -2 for wall and -3 for doors
  __global int *dId,     // direction flag form patches
  __global float *S,     // static floor field
  __global float *D,     // dynamic floor field
  __global float *Adepts_prob,// adepts success probability

  // constants - model environment
  const int ac,          // agents count
  const int r,           // agent vision radius
  const int width,       // world width - in patches (int)
  const int height,      // world height - in patches (int)
  const int min_px,      // min px coordinate
  const int max_px,      // max px coordinate
  const int min_py,      // min py coordinate
  const int max_py,      // max py coordinate
  // constants - model parameters
  const float Ka,
  const float Ks,
  const float Kd,
  const float Ki

){
  int id = get_global_id(0);
  bool bProcessKernel = (id < ac && 1 != Dead[id]);
  int start_adepts;         // index into array of adepts
  int adept_index;          // index into array of adepts - for this particular agent
  int allowed_to_move = 0;  // not yet allowed to move
  int want_to_move = 0;     // not yet decided to move
  int pi;

  // next patch index and cooredinates
  int npi;
  int npx;
  int npy;
  int pdx;
  int pdy;
  int my_winner_dir;

  if(bProcessKernel) {
    // --------------------------------------------------------------------------------------------------
    // STEP 2 - Prepare move -> register to adepts array where agent wants to move
    // --------------------------------------------------------------------------------------------------
```

```
pi = (PY[id] - min_py) * width + PX[id] - min_px;
allowed_to_move = 0; // not yet allowed to move
float p[6] = {0};
float a[6] = {0};
float N = 0;
int dir_i;
float max_a = -1;
int max_dd = -1;

for(dir_i = 0; dir_i < 6; dir_i++) {
  pdx = dirs[dId[pi]][dir_i][0];
  pdy = dirs[dId[pi]][dir_i][1];
  npx = PX[id] + pdx;
  npy = PY[id] + pdy;
  npi = (npy - min_py) * width + npx - min_px;

  if (IS_IN_RANGE(npx, npy)) {
    float counter = 0;
    int i = 0;
    int next_cell;

    while (i < r && IS_IN_RANGE(npx, npy)) {
      next_cell = (npy - min_py) * width + npx - min_px;
      if (CONTENT_WALL == C[next_cell]) {
        // next cells are considered as obstacles
        counter += (r - i);
        break;
      }
      if (CONTENT_DOORS == C[next_cell]) {
        // next cells are considered as empty
        break;
      }
      if (C[next_cell] >= 0) {
        // any pedestrian is here - increase counter
        counter++;
      }
      pdx = dirs[dId[next_cell]][dir_i][0];
      pdy = dirs[dId[next_cell]][dir_i][1];
      npx += pdx;
      npy += pdy;
      i++;
    }

    // calculate transition probability measure
    a[dir_i] = Ka * (1 - ((float)counter / r)) * exp(Ks * S[npi]) * exp(Kd * D[npi])
        * exp(Ki * ((last_dir[id] == dir_i)?1:0));
    N += a[dir_i];
  }
}

// normalize transition probabilities
if (N > 0) {
  for(dir_i = 0; dir_i < 6; dir_i++){
    p[dir_i] = a[dir_i] / N;
  }
}

// get winner
float max_p = 0;
int max_p_dir = -1;
my_winner_dir = -1; // not yet decided

// DESIRED solution - decide by probability distribution
uint2 rvec = (uint2)(RND1[id], RND2[id]);
float random_result = rand_float(&rvec);
RND1[id] = rvec.x;
RND2[id] = rvec.y;
float p_inc = 0;
for(dir_i = 0; dir_i < 6; dir_i++){
  p_inc += p[dir_i];
  if (p_inc > random_result) {
    my_winner_dir = dir_i;
    break;
  }
}
if (my_winner_dir != -1) {
      // winner patch
  pdx = dirs[dId[pi]][my_winner_dir][0];
  pdy = dirs[dId[pi]][my_winner_dir][1];
  npx = PX[id] + pdx;
  npy = PY[id] + pdy;

  if (IS_IN_RANGE(npx, npy)) {
    npi = (npy - min_py) * width + npx - min_px;
    if (CONTENT_EMPTY == C[npi] || CONTENT_DOORS == C[npi]) {
```

```
        // register this agent as wanting to step into the patch
        start_adepts = npi * 6;
        adept_index = npi * 6 + my_winner_dir;
        uint2 rvec = (uint2)(RND1[id], RND2[id]);
        Adepts_prob[adept_index] = rand_float(&rvec);
        RND1[id] = rvec.x;
        RND2[id] = rvec.y;
        want_to_move = 1;
      }
    }
  }
}

// ***********************************************************************************************************
barrier(CLK_GLOBAL_MEM_FENCE);
// ***********************************************************************************************************

if(bProcessKernel) {
  // -------------------------------------------------------------------------------------------------------
  // STEP 2 - decide the winner of adepts
  // -------------------------------------------------------------------------------------------------------
  if (1 == want_to_move) {
    int winner_adept = -1;
    float winner_prob = -1;
    for(int i = start_adepts; i < (start_adepts + 6); i++) {
      if (Adepts_prob[i] > winner_prob) {
        winner_prob = Adepts_prob[i];
        winner_adept = i;
      }
    }

    if (winner_prob != -1 && (winner_adept - start_adepts) == my_winner_dir) {
      // The current agent is the winner - do the move !!!              last_dir[id] = my_winner_dir;

      if (CONTENT_DOORS == C[npi]) {
        Dead[id] = 1;
      } else {
        C[npi] = id;
      }

      // new patch
      PX[id] = npx;
      PY[id] = npy;
      // and nex absolut coordinates
      X[id] += pdx;
      Y[id] += pdy;

      // release my former patch
      C[pi] = -1;
    }
  }
}

// ***********************************************************************************************************
barrier(CLK_GLOBAL_MEM_FENCE);
// ***********************************************************************************************************

if(bProcessKernel) {
// -------------------------------------------------------------------------------------------------------
// STEP 3 - clear adepts registration to make it ready for the next kernel run
// -------------------------------------------------------------------------------------------------------
  if (1 == want_to_move) {
    Adepts_prob[adept_index] = -1;
  }
}
}
//---------------------------------------- END OF KERNEL FUNCTION ----------------------------------
```

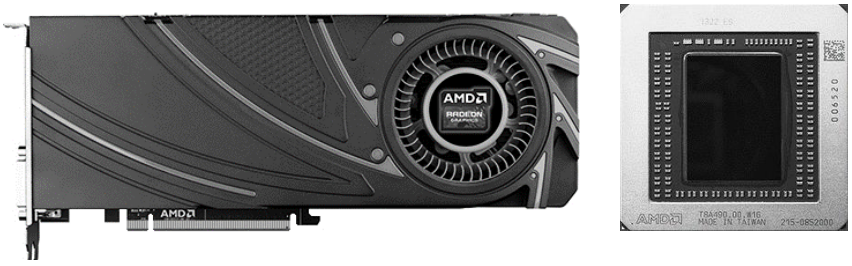
Příloha C – Použité experimentální prostředí

Pro vývoj NL2OCL a modelů, které toto rozšíření používají, stejně jako pro provedení experimentů nad těmito modely (viz kap. 6.1, 6.2 a 6.3), byla použita konfigurace standardního PC s těmito´parametry:

- CPU Intel i7-4790K, 4.00 Ghz, RAM 16GB, SSD: 500GB, OS: Windows 10, 64-bit

Použitá grafická karta: AMD Radeon R9 290X, parametry karty jsou následující:

- Výrobní technologie: 28 nm - Výrobní technologie určuje hustotu integrace čipu a přímo ovlivňuje spotřebu čipu v zátěži a tím i ekonomičnost provozu.
- Počet tranzistorů: 6,2 milionů - Počet tranzistorů vychází z výrobní technologie a velikosti plochy čipu, která je u R9 438 mm², určuje výpočetní schopnosti čipu.
- Paměť: 4 GB, šířka paměťové sběrnice: 512 bitů, celková propustnost: 320 GB/s - Velikost interní paměti grafické karty udává, jak velké úlohy je možné grafickým čipem najednou realizovat. Šířka sběrnice a celková propustnost paměťové sběrnice určují s jakou latencí způsobenou paměťovými operacemi budou výpočty na grafické kartě probíhat. Při paralelizaci výpočtů je zpoždění způsobené operacemi s pamětí nejkritičtějším místem celkového výkonu.
- Pracovní frekvence grafického jádra: 1GHz, taktovací frekvence paměti:1250 MHz - Taktovací frekvence určuje, s jakou rychlostí jsou prováděny instrukce na výpočetních jednotkách grafické karty.
- Počet výpočetních jader: 44, počet shading units: 2816 - Výpočetní jádro je autonomní výpočetní jednotka, která dokáže provádět výpočty nezávisle na ostatních výpočetních jednotkách.
- Výpočetní výkon: 5,632 GFLOPS - Celkový výpočetní výkon grafické karty – počet operací v plovoucí řádové čárce, které je grafická karta schopná provést za 1 sekundu.
- Podpora OpenCL: verze 2.1 - Od této verze se odvíjí, jaké OpenCL API je možné použít pro ovládání OpenCL platformy a zařízení a pro programování OpenCL kernelů.



Obrázek 43 - GPU AMD Radeon R9 290X s grafickým čipem Hawai, zdroj: (Techpowerup, 2017)