VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

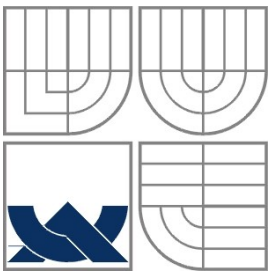
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SIMULÁTOR UMĚLÉHO ŽIVOTA

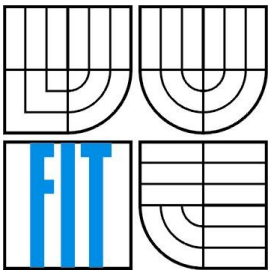
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Radim Luža
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SIMULÁTOR UMĚLÉHO ŽIVOTA

PARALLEL SIMULATOR OF ARTIFICAL LIFE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — Radim Luža
AUTHOR

VEDOUCÍ PRÁCE — Ing. David Martinek
SUPERVISOR

BRNO 2007

Abstrakt

Práce popisuje projekt paralelního simulátoru agentního systému. Stručně uvádí teorii nutnou k pochopení problematiky, dále se zabývá návrhem takového simulátoru a paralelismem ve zpracování a na závěr rozebírá výsledky dosažené implementovanou aplikací. Přináší srovnání použitých algoritmů na výpočetních systémech různých parametrů.

Klíčová slova

Agentní systém, umělá inteligence, simulace, MAS, paralelní

Abstract

This thesis describes project of parallel simulator of agent system. At the begining it explains necessary theory. Further follows description of design of particular software application. Final part contains set of test results and discussion. In the final part there is also comparison of parallel algorithms used for next simulator state computation.

Keywords

Agent system, artificial intelligence, simulation, MAS, parallel

Citace

Radim Luža: Paralelní simulátor umělého života, bakalářská práce, Brno, FIT VUT v Brně, 2009

Paralelní simulátor umělého života

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Davida Martinka. Další informace mi poskytli Ing. Martin Hrubý, Ph.D. a Doc. Ing. František Vítězslav Zbořil, Csc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Radim Luža
20. května 2009

Poděkování

Tímto bych rád poděkoval panu inženýrovi Davidovi Martinkovi, který mě vedl během zpracovávání této práce a jehož podpora a rady výrazně přispěly ke zvýšení kvality této práce. Dále bych chtěl poděkovat Dr. Ing. Petrovi Peringerovi, správci výpočetních serverů, který mi umožnil testování programu za účelem analýzy výsledků.

Obsah

1 Úvod

Vážený čtenáři, prostřednictví této bakalářské práce bych vás rád uvedl do problematiky umělého života a představil vám svůj projekt simulátoru umělého života. Ačkoliv je v rámci práce implementován simulátor, není simulace samotná hlavním cílem práce. V této práci se zaměřuji především na využití souběžného výpočtu ve více nezávislých vláknech za účelem dosažení výsledků v kratším čase – tzv. paralelizace. Simulátor samotný tak vytyčuje oblast paralelizace, kterou se tato práce zabývá. Hlavními cíli je vyzkoušení různých přístupů k paralelizaci, srovnání těchto přístupů z hlediska výkonu, porovnání přínosu paralelizace na různých výpočetních systémech a vyvození závěru z těchto experimentů.

Pro pochopení toho, o čem tato práce pojednává je dobré vysvětlit pojmy tvořící samotný název práce. Jsou to pojmy simulátor a umělý život. Simulátor je prostředek pro realizaci experimentů se simulačními modely. Simulační model je model původního reálného systému, se kterým lze provádět simulační experimenty. Simulátory obecně mohou mít více zaměření, v našem případě se ale bude jednat o program simulující jevy umělého života ve virtuálním světě.

Pojem umělý život je poměrně široký. Obecně zahrnuje veškeré formy napodobování projevů života stroji a technickými a matematickými prostředky. V této práci jsem se však zaměřil na úzký profil umělého života – zkoumání chování a spolupráce umělých entit pomocí tzv. agentního systému (pojem podrobněji vysvětlen v kapitole 2.1).

Důležitým aspektem hodnocení každé technologie je její uplatnění v praxi. Právě agentní systémy našly a stále nachází poměrně významná uplatnění především v řešení úloh umělé inteligence. Například problém obchodního cestujícího je možné poměrně efektivně řešit pomocí tzv. mravenčí kolonie, což je typický příklad agentního systému. Za tímto účelem byl přímo navržen algoritmus ACS (více o něm v kapitole 3.3). Dalším nasazením algoritmu na bázi agentního systému je algoritmus NetAnt využívaný pro řízení síťového směrování.

Celá práce je tvořena sedmi kapitolami včetně tohoto úvodu. Kapitola s názvem *Pojmy a technologie použité v práci* popisuje podrobněji teorii okolo agentních systémů. Zaměřuje se také na programové prostředky použitelné a případně použité při implementaci simulátoru. Kapitola *Paralelismus* se detailněji zabývá problematikou paralelního zpracování. Uvádí vývojové prostředky pro implementaci paralelismu a zmiňuje projekty s podobným zaměřením jako tato práce. Kapitola *Návrh implementace* se zabývá podrobněji fází implementace. Popisuje mj. problémy, které se při implementaci vyskytly, a jejich řešení. V kapitole *Implementace* je zjednodušeně zdokumentována implementace programové části práce. Kapitola *Testování* pak popisuje testování vlastností implementovaného simulátoru a porovnává různá řešení z implementační fáze. Poslední kapitola *Závěr* je pak celkovým zhodnocením práce. Hodnotí dosažené výsledky vůči jednovláknovému přístupu. Zabývá se také možnostmi rozšíření simulátoru.

2 Pojmy a technologie použité v práci

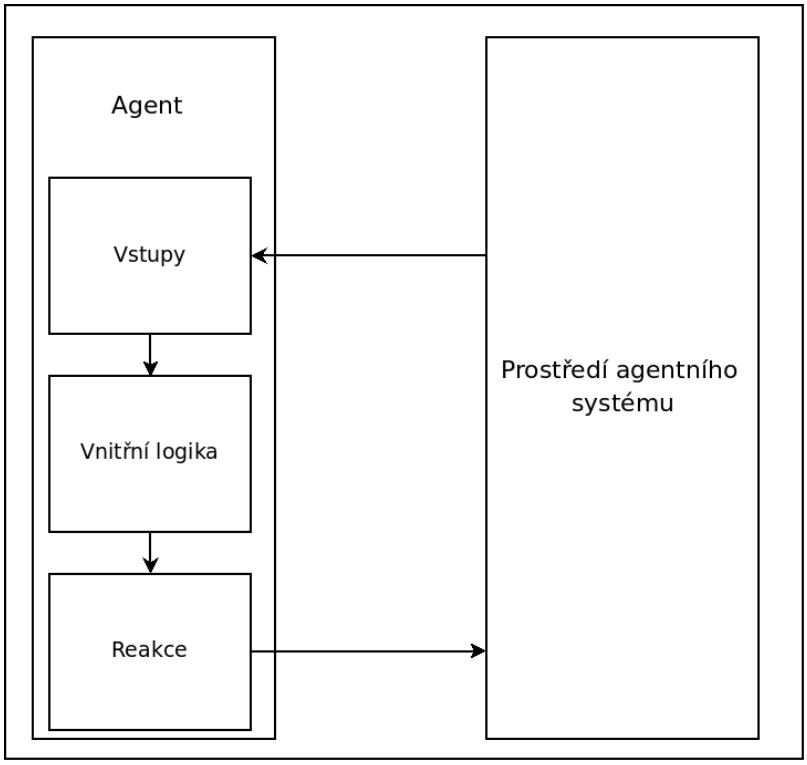
Tato kapitola popisuje teoretickou část práce a technické prostředky využité při implementaci z pohledu vlastností a licencí. Vysvětluje také používané pojmy.

2.1 Agentní systém

Agentní systém je systém založený na aktivitě autonomních prvků – agentů. Agentní systém je složený z prostředí a obecně nějakého počtu agentů pohybujících se v tomto prostředí. Prostředí může být dvourozměrné, vícerozměrné, obecný graf, může být pouze pasivní, nebo může být realizováno jako aktivní celulární automat. V této realizaci je prostředí dvourozměrná pasivní mapa – čtvercová síť. Na této síti se pohybují agenti schopni mimo jiné toto prostředí modifikovat. Podobně jako jiné významné pojmy týkající se agentních systému je agent popsán níže.

2.1.1 Agent

Agent v této souvislosti je chápán jako samostatná entita v agentním systému schopná samostatného rozhodování a nezávislého chování. Účelem agenta je dosažení určitého cíle. Tomuto cíli agent podřizuje své chování. Agent reaguje na vstupy ze světa agentního systému a svými výstupy provádí reakce na tyto vstupy. Jádrem agenta je jeho vnitřní logika. Ta může být různě složitá – od nejjednodušších sad podmínek, přes logiku uchovávající vnitřní stav, až po komplexní umělou inteligenci schopnou učení a přizpůsobování se. Princip agenta je popsán na obrázku 2.1.



Ilustrace 2.1: Princip agenta – předloha viz. literatura [1]

Detailněji a velmi podrobě se vlastnostmi a rysy agentů zabývá publikace *Hidden Order* [2]. Typickým nasazením agentů je spolupráce více agentů za účelem dosažení cíle. Takovýto agentní systém je pak označován jako MAS – Multi Agent System.

2.1.2 MAS

MAS neboli Multi Agent System je agentní systém, v němž se vyskytuje více navzájem spolupracujících agentů. Typickým příkladem takovýchto agentních systémů jsou nejrůznější kolonie živočichů a různá společenství. MAS byly poprvé v praxi nasazeny za účelem řízení a optimalizace letecké dopravy. Podrobnější informace lze nalézt v odkazované literatuře [3]. Navrhovaný simulátor musí být schopen realizovat právě MAS stejně jako agentní systém s jedním agentem (například procházení bludiště).

2.1.3 Emergence

Emergence je jev, při kterém dochází ke vzniku nových vlastností systému jako celku na základě interakce jeho komponent. Emergence se vyskytuje i v běžné realitě okolo nás – nejedná se tedy o laboratorní záležitost. Příkladem může být kolonie mravenců při hledání potravy. Mravenec, který potravu najde, uchopí část, kterou unese, a vrací se s ní k mraveništi. Za sebou nechává feromonovou stopu. Když na tuto stopu narazí další mravenci, postupují po ní za potravou a při návratu stopu posilují. V důsledku tohoto chování je schopna mravenčí kolonie jako celek prohledávat své okolí a při nalezení potravy tuto efektivně odnést do mraveniště. Právě MAS se využívají pro zkoumání a optimalizaci jevů emergence. Další příklady emergence a širší vysvětlení tohoto pojmu viz. [3].

3 Paralelismus

3.1 Prostředky pro paralelizaci

Prostředky pro paralelizaci jsou zde myšleny softwarové prostředky, které umožňují paralelní provádění algoritmu a případně zjednodušují návrh paralelního provádění a zpřehledňují paralelismus v kódu a výsledném programu.

Prostředků splňujících tyto požadavky je poměrně hodně. V rámci této práce jsou podrobněji popsány následující: systémové knihovny [4] – prostředky, které jsou přímo součástí operačních systémů a jejich API je podmnožinou rozhraní systémových knihoven, knihovna TBB [5] – tato knihovna je v práci rozebrána podrobněji, neboť byla využita při implementaci, knihovna OpenMP [6] – navržená a udržovaná skupinou předních výrobců hardwaru a softwaru a na závěr knihovna ParallelFX [7] – rozšíření frameworku .NET.

3.1.1 Systémové knihovny

Prvním zmíněným prostředkem pro paralelizaci jsou systémové knihovny. Tyto sice neposkytují abstrakci nad paralelismem v kódu, ale jsou zcela nezbytné pro samotné umožnění paralelismu na dané platformě a knihovny vyšší úrovně na nich staví.

Hlavní nevýhodou systémových knihoven je právě závislost na konkrétní platformě. Existují sice sjednocující standardy, jako POSIX Threads, ale přesto mají nejčastěji používané operační systémy současnosti (MS Windows, MAC OS, Linux) rozdílná rozhraní pro využití paralelismu aplikačním programátorem. Přesto tato rozhraní musí poskytovat podobnou funkcionalitu. Tato funkcionalita zahrnuje vytvoření paralelního vlákna v algoritmu (vlákno je zde chápáno jako obecný pojem označující souběžnou větev algoritmu), zrušení paralelního vlákna, řízení synchronizace současného běhu vláken a řízení souběžného přístupu k datům. Jako příklad rozhraní systémové knihovny jsem zvolil právě POSIX Threads – viz. [4], neboť toto rozhraní je implementováno ve více operačních systémech (Linux, různé větve BSD a existuje i implementace pro MS Windows). Pokud je dále v práci zmiňováno API, jedná se – pokud není řečeno jinak – o API pro programovací jazyk C/C++.

Životní cyklus každého vlákna začíná jeho vytvořením. Vytvořením vlákna je alokována paměť a zavolána funkce představující tělo vlákna. Typicky bývá vytvořené vlákno aktivní a může být ihned spuštěno v závislosti na systémovém plánovači. V POSIX Threads vytvoření vlákna zajišťuje funkce *pthread_create*. Jako parametr funkce vyžaduje mimo jiné ukazatel na funkci, která se stane tělem vlákna. Spuštěné vlákno je prováděno souběžně s rodičovským vláknem ostatními vlákny daného procesu, dokud jeho běh neskončí. Ukončení vlákna nastává buď dokončením funkce vlákna a návratem z ní, nebo explicitním přerušením vlákna funkcí *pthread_exit*, která jako parametr vyžaduje ukazatel na data, která se předají připojenému vláknu.

Výše zmíněné připojení vlákna je jeden z prostředků synchronizace současného běhu vláken. Operace *join* - připojení představuje propojení volajícího vlákna s jiným vláknem takovým způsobem, že volající vlákno nebude ukončeno dříve, než se ukončí vlákno připojené. Této vlastnosti lze využít například při návratu hodnoty ze synovského vlákna, a nebo při čekání na dokončení více synovských vláken. Operaci *join* v POSIX Threads zajišťuje funkce *pthread_join*, která jako parametr bere identifikátor vlákna, se kterým má být volající vlákno spojeno. Opakem *join* je operace *detach* - odpojení, která naopak zajišťuje, že při ukončení požadovaného vlákna budou okamžitě dealokovány prostředky tomuto vláknu přidělené a že ukončení toho vlákna neovlivní běh vláken ostatních. POSIX Threads funkce *pthread_detach* zajišťující tuto funkcionalitu vyžaduje jako parametr identifikátor vlákna, které má být odpojeno.

Posledním nutnou podmínkou podpory paralelismu je řízení souběžného přístupu k datům. Vzhledem k tomu, že vlákna běžící souběžně jsou řízena systémovým plánovačem, nelze zpravidla říci, kdy bude které vlákno přistupovat k datům. Pokud více vláken přistupuje ke stejným datům, může dojít ke kolizi, kdy například jedno vlákno se pokusí číst data, která mu však během čtení jiné

vlákno přepisuje, tudíž první vlákno získá nekonzistentní hodnotu. Takto ohrožené prostředky se nazývají kritické sekce. Aby se kolizním jevům předešlo, musí knihovna nabízet prostředky pro výlučný přístup k těmto kritickým sekcím.

Základem tokového výlučného přístupu je *mutex* - vzájemné vyloučení. *Mutex* představuje zámek, který může být uzamčen a pokusy o jeho znovu-uzamčení jsou odloženy, nebo se vrátí s neúspěchem, dokud tento zámek není opět uvolněn. Vlákno se před vstupem do kritické sekce pokusí zámek uzamknout. Pokud je zámek volný, vlákno jej uzamkne a pokračuje přístupem do kritické sekce. V opačném případě je buďto odloženo a čeká, kdy se tento zámek uvolní, nebo pokračuje dále s informací, že kritická sekce je momentálně nedostupná. Vytvoření mutexu zajišťují funkce *pthread_mutex_init* nad datovou strukturou *pthread_mutex_t* a zrušení mutexu funkce *pthread_mutex_destroy*. Obě funkce vyžadují ukazatel na mutex, s nímž pracují. Zamykání a odemykání mutexu potom zajišťuje trojice funkcí *pthread_mutex_lock*, *pthread_mutex_trylock* a *pthread_mutex_unlock*. Funkce s *lock* v názvu zamykají mutex, funkce *pthread_mutex_unlock* pak mutex odemyká a uvolňuje pro další pokusy o uzamčení. Zatímco funkce *pthread_mutex_lock* v případě neúspěchu přeruší běh volajícího vlákna a vyčká na uvolnění mutexu, funkce *pthread_mutex_trylock* v případě neúspěchu skončí okamžitě s návratovou hodnotou značící neúspěšný pokus o uzamčení. Těmito prostředky lze provést základní řízení přístupu k datům. V rámci systémových knihoven existují i složitější a specifičtější prostředky, ale ty už nemusí být ve všech knihovnách implementovány a jejich využití lze většinou nahradit právě mutexy, a proto nejsou v této práci dále popisovány.

Využití systémových knihoven pro implementaci paralelismu v aplikačních programech je poměrně často používané, avšak má svá omezení. Kromě výše zmíněné nepřenositelnosti mezi různými operačními systémy bývá hlavním hlavním argumentem proti využití systémových knihoven poměrně složitá implementace rozsáhlejších paralelních řešení.

3.1.2 Knihovna TBB

Knihovna TBB (Threading Building Blocks) je knihovna zjednodušující paralelizaci vyvíjená firmou Intel. Knihovna je multiplatformní a abstrahuje správu procesů a vláken v podporovaných operačních systémech. Řeší také rozkládání zátěže na více procesorů systému. Distribuce zátěže mezi procesory či procesorová jádra může být knihovnou řešena zcela transparentně bez explicitního řízení. Umožňuje tak zjednodušit práci aplikačního programátora na zápis vhodných paralelizovatelných funkcí a následné volání prostředků knihovny pro pro realizaci výpočtu. Řízení paralelismu jako takové – vytváření, rušení a synchronizace výpočetních vláken – je pak zajištěno knihovnou TBB. Tím se zásadně liší od předchozích systémových knihoven, kde tyto operace musel řídit při programování programátor aplikace.

Knihovna TBB je komplexní nástroj poskytující poměrně širokou škálu paralelizačních prostředků. Z hlediska této práce se zaměřím na dvě oblasti těchto prostředků: prostředky pro řízení paralelního výpočtu a abstraktní datové typy s podporou paralelismu. Toto dělení bude použito i při rozboru následujících knihoven. Mimo tyto oblasti poskytuje knihovna TBB i prostředky pro řízení řízení průběhu výpočtu realizovaného prostředky knihovny, na platformně nezávislé časovače a prostředky ekvivalentní systémovým knihovnám, bez závislosti na konkrétní platformně. Užitečnou vlastností knihovny je také možnost využití systémových knihoven na dané platformně bez nebezpečí kolize s prostředky knihovny TBB.

Na vyšší úrovni abstrakce jsou v TBB k dispozici prostředky, kterými lze obecně vytknout úsek kódu a ten potom provést paralelně. Na kolik vláken se při výpočtu proces rozvětví určují tzv. *partiotionery*. *Partiotioner* funguje jako funkce, která podle zadané granularity, předchozího stavu a vlastností konkrétního systému vrací množinu vláken, množinu datových podmnožin a informaci o tom, kterému vláknu bude která datová podmnožina přidělena. Granularita určuje, kolik prvků z množiny, nad kterou je prováděn paralelní výpočet, bude zpracováváno v rámci jednoho vlákna. Správná granularita je klíčová pro efektivní výpočet. Příliš nízká granularita (příliš velký počet prvků v jednom vláknu) způsobí, že počítač není plně využit, naopak příliš vysoká granularita způsobí, že režie paralelismu převýší časovou výhodu získanou jeho nasazením. Proto TBB nabízí možnost na-

stavovat granularitu jak ručně, tak automaticky, aby bylo možno využít optimální hodnoty. *Partitionery* jsou v knihovně tři: *simple_partiotioner*, *auto_partiotioner* a *affinity_partiotioner*.

Simple_partiotioner dělí množinu prvků na paralelní podmnožiny, dokud je dělitelná. Pro tento partitioner je kritický atribut granularity, který určí mez, za kterou nelze množinu dále dělit. Nejmenší možná hodnota je *1*, kdy se množina rozdělí na jednotlivé prvky. *Auto_partiotioner* dělí množinu prvků automaticky pokud možno vhodně. Zadaná granularita určuje spodní omezení tohoto automatického dělení, nicméně její hodnota není tak kritická jako u pariotioneru předchozího. Poslední z možných je *affinity_partiotioner*. Tento optimalizuje paralelní výpočet zefektivněním přístupu do paměti. Snaží se přiřazovat jednotlivým vláknům iterátory k datům tak, aby byly minimalizovány výpadky TLB.

Nejrozsáhlejší částí knihovny jsou prostředky pro určení způsobu a oblasti paralelizace. Jsou to třídy, které abstrahují jednotlivé přístupy k paralelizaci. Patří sem *parallel_for*, *parallel_reduce*, *parallel_scan*, *parallel_do*, *pipeline* a *parallel_sort*.

Parallel_for je prostředek pro paralelizaci cyklů. Nutnou podmínkou pro využití tohoto prostředku je, že iterace cyklu musí být na sobě nezávislé (nelze využívat výsledky minulých iterací) a nesmí docházet ke kolizi v přístupu k datům. Data musí být buďto na sobě nezávislá pro jednotlivé iterace a nebo musí být synchronizačními prostředky zajištěno, aby ke koliznímu přístupu nedocházelo.

Parallel_reduce slouží právě k realizaci výpočtu, kde výsledek je ovlivněn všemi iteracemi cyklu (typický příklad je součet prvků pole). Tato třída umožňuje zpracovat mezivýsledky jednotlivých vláken po jejich dokončení. V ostatních ohledech je průběh výpočtu víceméně shodný s *parallel_for*.

Parallel_scan je prostředek umožňující paralelizaci sériového výpočtu. Jedná se o sériový výpočet nad množinou prvků, kde jednotlivé iterace jsou spolu vzájemně propojeny tak, že v dílčím výsledku iterace figurují dílčí výsledky předchozích iterací v asociativní operaci. Příkladem takového výpočtu je například výpočet Fibonaccioho řady. Takto paralelizovaný výpočet je sice celkově náročnější, ale lze ho rozložit na více vláken, která lze vykonat paralelně a tak dosáhnout výsledku rychleji. Problematika paralelizace sériového výpočtu je však mimo rozsah této práce.

Prostředek *parallel_do* umožňuje provádět operaci nad množinou dat, jejíž velikost není předem známá. V průběhu zpracovávání dat je možno další data přidávat. Pro zpracování dat jinak platí stejná omezení, jako u *parallel_for*.

Pipeline je prostředek umožňující vytvoření soustavy funkcí, kterými postupně prochází data a každá z těchto funkcí provádí dílčí zpracování (analogie s výrobní linkou). *Pipeline* se pak stará a optimální paralelní provedení těchto funkcí tak, aby výsledek odpovídal zadefinovanému zřetězenému zpracování.

Posledním z abstraktních paralelizačních prostředků v TBB je *parallel_sort*. *Parallel_sort* umožňuje paralelizovat proces řazení datové množiny na základě zadané komparační funkce. Výhodou využití *parallel_sort* je, že paralelizaci řídí transparentně, nevýhodou je pak to, že řazení s využitím tohoto prostředku je nestabilní.

Další částí knihovny významnou především z hlediska zjednodušení bezkolizního přístupu k datům ze souběžných vláken jsou již zmíněné abstraktní datové typy s podporou paralelismu. Tyto datové typy zajišťují bezpečný souběžný přístup ke komplexnějším datovým strukturám. Jejich výhodou oproti použití výlučného přístupu k takové datové struktuře je jemnější systém řízení přístupu, který umožňuje přístup k datům zefektivnit a v mnoha ohledech oproti výlučnému přístupu paralelizovat. Například při modifikaci jedné položky v datovým typu na principu STL vektoru je možné souběžně číst obsah jiné položky, aniž by tím hrozila kolize v přístupu k datům. TBB nabízí tyto kontejnery pro tvorbu datových typů: *concurent_hash_map*, *concurent_queue* a *concurent_vector*.

Kontejner *concurent_hash_map* je hash tabulka bezpečná pro souběžný přístup. Data jsou v tomto kontejneru řazena zadanou hashovací funkcí a jsou uspořádána do stromu podobně jako u STL kontejneru *map*. Položky v kontejneru je možno vyhledávat na základě klíče. Prvek kontejneru představuje pár párech klíč-data.

C*oncurent_queue* je kontejner modelující frontu (FIFO) bezpečnou pro souběžný přístup. Prvky lze vkládat na konec a vyjímat ze začátku fronty. Pro účely ladění poskytuje *concurent_queue* i možnost projít všechny prvky ve frontě.

Třetí a poslední kontejner s bezpečným souběžným přístupem je *concurent_vector*. Kontejner je analogií STL kontejneru *vector*. Chová se jako dynamické pole – přístup k datům je sekvenční. V souvislosti se souběžným přístupem k datům je rovněž užitečné zmínit, že STL kontejnery v C++ jsou bezpečné pro souběžné čtení a výlučný přístup je nutný jen v případě zápisu. Této skutečnosti je s výhodou využíváno v implementaci programové části práce.

Významným a mnohdy rozhodujícím aspektem každé knihovny je její licence. Knihovna TBB je licencována pod dvojí licencí. Jedna licence je proprietární licence firmy Intel a druhá je licence GPL. Licence GPL ve zkratce umožňuje přístup ke zdrojovému kódu knihovny a umožňuje její neomezené použití za předpokladu, že projekty, které ji využívají budou šířeny pod licencí GPL nebo kompatibilní. Licence GPL v podstatě vynucuje otevření zdrojového kódu projektu, který knihovnu použije, na druhou stranu však umožňuje bezproblémové nasazení, studium, integraci i modifikaci takto licencovaných projektů na úrovni zdrojového kódu. Nepřímým, ale ne nepodstatným důsledkem této politiky je fakt, že knihovna je přímo obsažena v repositářích mnoha linuxových distribucí (například Ubuntu 8.10 – knihovna ve starší verzi 2.0r020-1), a tak je její instalace a konfigurace snadná a automatizovaná. Právě multiplatformnost, srozumitelné rozhraní a vhodná licence byly hlavními důvody k použití této knihovny v implementaci. Více informací lze nalézt v položce [5] použité literatury – Intel Threading Building Blocks 2.1 for Open Source.

3.1.3 OpenMP

OpenMP je paralelizační knihovna vyvíjená neziskovou korporací ARB. ARB zahrnuje zástupce velkých firem, jako IBM, Intel, HP, NEC, Mifrosoft a jiné, ale i organizací jako NASA. ARB si v projektu OpenMP klade za cíl vytvořit škálovatelnou portabilní knihovnu pro paralelní výpočty se sdílenou pamětí. OpenMP nabízí rozhraní pro jazyky C a Fortran.

Oproti předchozí TBB OpenMP nenabízí abstraktní datové typy s podporou paralelismu. Obsahuje prostředky pro řízení paralelního výpočtu, prostředky pro řízení přístupu k paměti a synchronizační prostředek mutex. Nejvýznamnější část OpenMP představují právě prostředky pro řízení paralelního výpočtu, proto jim bude věnována nejdelší část popisu.

OpenMP nepoužívá na rozdíl od TBB třídy, ale direktivy preprocesoru *#pragma*. Těmito direktivami je potom blokům kódu přiřazováno určité chování. Každá direktiva náležící OpenMP začíná řetězcem *#pragma omp* a pokračuje označením prostředku řízení výpočtu a klauzulí definující správu paměti pro daný blok. Celý tvar direktivy je tedy *#pragma omp název_prostředku [klauzule]*. Klauzule definující správu paměti je nepovinná. Následuje popis jednotlivých direktiv.

Direktiva #pragma omp parallel vytvoří několik vláken a zahájí paralelní provádění označeného bloku. Následující direktivy se musí vyskytovat v bloku označeném touto direktivou. Výjimku tvoří pouze kombinované direktivy – viz. níže. Direktiva *#pragma omp for* reprezentuje konstrukci paralelního *for* cyklu. Pro tělo cyklu musí být tradičně zajištěno, aby nedocházelo ke kolizím jednotlivých iterací. Direktiva *#pragma omp sections* vytvoří sadu sekcí, které se provedou paralelně. V bloku označeném touto direktivou je pak nutné direktivami *#parallel omp section* jednotlivé nezávislé bloky. Řetězec *#pragma omp single* je direktivou označující blok, který se v rámci paralelního zpracování provede jedno-vláknově. Direktiva *#pragma omp task* vytváří samostatnou úlohu, kterou lze provést paralelně se zbytkem výpočtu. Provádění paralelního výpočtu je synchronizováno koncem sekce, v níž jsou direktivy vyvolány.

Výjimku v této synchronizaci tvoří direktiva *#pragma omp master*, která označený blok provede jako hlavní vlákno skupiny paralelních vláken. Hlavní vlákno nemá implicitní vstupní a výstupní synchronizační body. Tyto synchronizační body lze explicitně vytvořit direktivou *#pragma omp barrier*, která přidává synchronizační bod, na němž se vlákna v rámci dané sekce zastaví a počkají na dokončení všech ostatních, případně direktivou *#pragma omp taskwait*, která způsobí, že konkrétní vlákno počká na dokončení všech synovských vláken.

Direktiva *#pragma omp atomic* zajišťuje atomické provedení označeného bloku. To znamená, že daný blok bude proveden v kuse bez přerušení, takže jiná vlákna nemohou narušit jeho průběh. Direktiva *#pragma omp flush* zajistí, že paměť vláken bude za běhu synchronizována s hlavní pamětí procesu. Veškeré změny se budou tedy okamžitě promítat do nadřazených bloků. To se týká jen proměnných touto direktivou označených. Direktiva *#pragma omp ordered* zajistí, že označený blok uvnitř cyklu bude proveden tak, jako by iterace cyklu byly prováděny sekvenčně. Direktiva *#pragma omp threadprivate* je potom opakem direktivy *#pragma omp flush* a způsobuje, že označené proměnné jsou zkopírovány lokálně pro vlákno a jejich původní obsah zůstává zachován. Jako poslední zmíním direktivy složené. Jedná se o *#pragma omp parallel for* a *#pragma omp parallel section*, které se chovají stejně jako direktivy *#pragma omp for* a *#pragma omp section*, ale není nutné uvádět je v bloku označeném *#pragma omp parallel*.

Druhou klíčovou částí OpenMP jsou klauzule definující přístup do paměti během provádění paralelního výpočtu. Klauzule přiřazují definované chování označeným proměnným. První klauzulí je *default(shared|none)*. Tato určuje chování proměnných referencovaných direktivami *#pragma omp parallel* a *#pragma omp task*. Určuje sdílení proměnných mezi úlohami. Klauzule *shared(seznam proměnných)* deklaruje explicitní seznam proměnných, které budou sdíleny mezi úlohami sekce paralelního výpočtu. Následuje sada klauzulí omezujících platnost proměnných pro konkrétní úlohu. Jsou to *private(seznam proměnných)*, která omezí platnost proměnných na konkrétní blok – jejich změny se neprojeví v nadřazených blocích, *firstprivate(seznam proměnných)*, která se chová stejně jako předchozí, ale navíc uvede před spuštěním úlohy proměnné do stavu, v jakém byly v místě, kde se nachází konstrukce bloku úlohy a nakonec *lastprivate(seznam proměnných)*, která učiní označené proměnné privátní pro úlohu, ale po dokončení úlohy promítne výsledný stav proměnných do nadřazeného bloku. Klauzule *raduction(operátor:seznam proměnných)* zajistí, že proměnné, které se lokálně změní během provádění jednotlivých úloh se po dokončení všech dílčích úloh sloučí do stejnojmenných nadřazených proměnných operátorem *operátor*. Poslední dvě klauzule řídí sdílení proměnných mezi úlohami. Jsou to *copyin(seznam proměnných)*, která zkopíruje hodnotu proměnných ze sekce *threadprivate* hlavního (master) vlákna do stejnojmenné sekce vláken synovských a klauzule *copyprivate(seznam proměnných)*, která šíří hodnotu *threadprivate* proměnných jedné úlohy jiné úloze. Takto ovlivněné proměnné jsou vyčteny seznamem proměnných klauzulí.

Na závěr je dobré ještě zmínit synchronizační prostředky OpenMP. Jak již bylo zmíněno výše, je jím pouze prostředek *mutex* (pomineme-li synchronizaci na základě direktiv). Pro řízení mutexu poskytuje knihovna sadu funkcí. Názvy funkcí začínají řetězcem *omp* a končí řetězcem *lock*. Další podrobnosti o knihovně OpenMP lze najít v použité literatuře [6].

3.1.4 ParallelFX

ParallelFX není přímo knihovna s rozhraním pro jazyk C/C++. Jedná se o nástavbu frameworku .NET vyvíjeného primárně firmou Microsoft. Poskytuje rozhraní pro jazyky podporované platformou .NET – tedy i C#. Framework .NET a s ním i nástavba ParallelFX jsou dostupné pouze pro operační systémy z rodiny MS Windows. Existuje však projekt MONO, který se snaží implementovat .NET i pro jiné platformy. V rámci tohoto projektu je portována i nástavba ParallelFX. Díky této komplikované a ne zcela kompletní portabilitě řešení na bázi .NET a ParallelFX a díky absenci rozhraní pro C++ (dostupné pouze rozhraní pro C#) je tato možnost zmíněna jen přehledově.

Sada prostředků pro řízení paralelizace v ParallelFX je v podstatě podmnožinou sady nabízené knihovnou TBB. ParallelFX poskytuje *Parallel.For* pro paralelizaci cyklů se zcela nezávislými iteracemi (analogie *parallel_for* v TBB), *Parallel.Agregate* pro paralelizaci cyklů, kde figuruje agregace dílčích výsledků jednotlivých iterací (analogie *parallel_reduce* v TBB) a *Parallel.Do*, což je prostředek umožňující spuštění souběžných podprocesů s tím, že návrat do hlavní větve nastane až ve chvíli, kdy jsou všechny podprocesy dokončeny. V tomto případě se jedná o funkcionalitu zcela odlišnou od *parallel_do* v TBB.

Abstraktní datové typy bezpečné pro souběžný přístup dostupně v ParallelFX jsou potom realizovány kontejnery *ConcurrentQueue* implementující frontu (FIFO) bezpečnou pro souběžný pří-

stup, *ConcurrentStack* implementující zásobník (LIFO) bezpečný pro souběžný přístup a *BlockingCollection*, což je fronta bezpečná pro souběžný přístup, která blokuje vlákna odebírající prvky z čela fronty do doby, kdy je možno tento požadavek odebrat. Pokud je fronta *BlockingCollection* prázdná, je odebírající proces pozastaven do doby, kdy dojde požadavek, který by mu mohl být předán. Do zvláštní kategorie potom patří kontejner *LazyInit*, který ačkoliv se jedná o datový kontejner, nezajišťuje bezpečný přístup k datům, ale oddaluje inicializaci zapouzdřeného objektu do doby, kdy je tento skutečně potřebný – tzv. lazy inicializace. Svou funkcí se tento kontejner tudíž řadí spíše mezi prostředky pro řízení paralelizace, než mezi prostředky zajišťující řízení přístupu k datům. Kromě tohoto výčtu obsahuje ParallelFX i jiné prostředky pro řízení přístupu k datům, jako jsou mutexy a semafory. Jejich popis už není v tomto stručném přehledu zahrnut, lze jej však najít spolu s rozšiřujícími informacemi o popsaných prostředcích v položce literatury [7].

3.2 Motivace

K zájmu na využití paralelizace v této formě simulace vede více skutečností. Jednoznačně hlavní motivací je možnost provést simulaci rychleji a dříve tak získat výsledky. Toto je důležité jednak v situacích, kdy na výsledky podmíněn další postup projektu, nebo kdy je účtován výpočetní čas, ale paralelizace může být někdy jediná možnost, jak dosáhnou běhu simulace v reálném čase.

Jiným a rovněž významným aspektem paralelizace je škálovatelnost výpočtu. Paralelní výpočetní vlákna jsou na sobě do jisté míry nezávislá a systémový plánovač je může řídit jednotlivě. V případě potřeby mohou být vlákna přesouvána mezi jádry a procesory, nebo případně některá i pozastavena ve chvílích, kdy je potřeba využít výpočetní výkon k jiným účelům. Škálovatelnost má význam i při přístupu k systémovým zdrojům. Zatímco jedno vlákno čeká na přidělení zdroje, jiné může běžet a využít dočasně volného času procesoru. Typickým případem pomalého systémového prostředku je pevný disk. Pokud simulátor intenzivně přistupuje na disk, může být paralelní zpracování oproti jednovláknovému rychlejší i na jedno-jádrových výpočetních systémech. V neposlední řadě je silnou motivací i touha experimentovat s paralelismem v aplikacích, což je téma, které ani dnes ještě není tak docela běžné.

3.3 Podobné projekty

Podobnými projekty jsou v této práci myšleny projekty zabývající se zefektivněním výpočtu simulace agentního systému pomocí paralelismu. Výběr podobných projektů není omezen jen na simulátory agentních systémů. Ačkoliv je paralelizace v simulaci poměrně hojně využívaným prostředkem, v případě agentních systémů je častější využití distribuovaného výpočtu. Distribuce výpočtu po síti se využívá především u rozsáhlých simulací.

Jedním z příkladů využití paralelismu v simulaci agentního systému je popsán v prezentaci firmy IBM Parallel Agent Based Simulation on PC Cluster [8]. Prezentace popisuje koncept simulátoru modelu Word of Mouth (simulace ústního šíření informace) s využitím prostředí Mozart Programming System. Jedná se o demonstrační aplikaci zmíněného prostředí. Mozart Programming System poskytuje prostředky pro paralelní i distribuovanou implementaci algoritmů umělé inteligence. Paralelizační prostředky Mozart Programming System jsou maximálně automatizované a pokud to je možné, zcela transparentní. Na tomto prostředí tedy pravděpodobně vznikly mnohé další podobné projekty, avšak z časových a rozsahových důvodů v této práci nejsou popsány.

Další případ využití paralelismu v agentním systému popisuje článek *Ant colony system: a cooperative learning approach to the travelingsalesman problem* [9]. Jedná se o využití paralelismu při řešení problému *obchodního cestujícího* (model) s využitím paralelismu. Ant colony system (ACS) je algoritmus založený na agentním systému, který v tomto případě slouží k nalezení optimální cesty při průchodu několika body rozmístěnými na ploše nebo v prostoru. Paralelismus je zde aplikován tak, že se dělí množina agentů v simulaci a jejich chování se provádí současně. Stejného principu využívá i simulátor agentních systémů implementovaný v rámci této práce.

4 Návrh implementace

Tato kapitola popisuje implementaci agentního systému tak, jak je realizována v rámci této bakalářské práce. Jsou zde popsány vstupy pro simulátor, uvedeny příklady vstupů, dále jsou zde popsány vnitřní struktura simulátoru, princip simulace, výstupy simulace a na závěr kapitoly je definována komunikace mezi vizualizačním klientem a samotným simulátorem.

Hlavní rysy návrhu jsou:

- Univerzálnost – simulátor je maximálně univerzální aby umožnil simulovat co nejširší okruh úloh řešených agentním systémem.
- Oddělená vizualizace – vizualizace není součástí simulace a při běhu simulace není třeba simulaci vizualizovat. Vizualizace je řešena samostatnou aplikací, která komunikuje se simulátorem po síti – může tedy běžet i na vzdáleném stroji.
- Paralelní provádění – simulátor by měl maximálně využívat výhod víceprocesorových systémů

4.1 Co je třeba definovat

4.1.1 Vstupy

- Definice simulace: odkud bude brána informace o mapě, odkud budou brány informace o agentech, kolik jakých agentů bude na mapě rozmístěno a kde, za jakých podmínek simulace končí a případně jak rychle má simulace probíhat – pro simulace, které pracují v reálném čase, kam ukládat výsledky simulace
- Definice mapy: jak je mapa velká, jak vypadá terén na této mapě z hlediska simulace, jak vypadá terén na mapě z hlediska vizualizace
- Definice agenta: inicializace výchozího stavu agenta, definice algoritmu chování agenta, definice vzhledu agenta ve vizualizaci, rozhraní agenta

4.1.2 Výstupy

- Okamžitý stav simulace: informace o simulačním čase, o stavu agentů, o okamžité podobě mapy
- Statistické informace: jak dlouho simulace běžela, kolik bylo provedeno změn na mapě, kolik zůstalo agentů a jakých po skončení simulace a případné další uživatelské statistické informace

4.1.3 Komunikace simulátoru s klientem vizualizace

- Získání grafický informací potřebných k vizualizaci: obrázky agentů, bitmapy pro ztvárnění povrchu mapy
- Získání informací o simulaci: rychlost simulace, popis simulace
- Získání okamžitých informací o simulaci: informace o agentech, jejich pozicích a vnitřních proměnných, informace o změnách na mapě
- Získání statistik simulace: doba běhu simulace, počet změn na mapě, uživatelsky definované statistické informace

4.2 Princip simulátoru

Simulátor je rozdělen na dvě části: samotný simulátor a klient vizualizace. Každá z těchto částí má vlastní řídicí algoritmus. Simulátor je pochopitelně výrazně složitější. Pro synchronizaci akcí agentů při simulaci byla zvolena technika vytváření fronty modifikujících požadavků. Toto řešení spočívá v tom, že veškeré akce, které mění stav simulátoru – přesun agenta, modifikace mapy, modifikace atributu agenta – jsou vyčleněny zvlášť do dočasné fronty. Provádí se až po tom, co je skončen poslední skript agenta. Fronta modifikujících akcí vzniká proto, aby všichni agenti měli stejné výchozí podmínky. Tím se dosáhne jevu dokonalého kvaziparalelismu – situace, kdy všechny akce agentů byly prováděny jakoby ve stejný okamžik a v neomezeně krátkém čase – tudíž se nijak neovlivňují. Odůvodnění použití tohoto řešení bude popsáno v diskusi k návrhu. Simulátor je diskrétní – simulace probíhá v krocích.

4.2.1 Algoritmus simulátoru – hlavní proces

1. Načtení konfiguračního souboru: přečtení konfiguračního souboru, jeho zpracování, získání případných dalších souborů
2. Inicializace simulace: inicializace mapy, inicializace statistických proměnných, rozmístění agentů na mapě, inicializace vnitřních proměnných agentů
3. Provedení akcí jednotlivých agentů v simulaci – požadavky, které modifikují atributy agentů nebo mapu jsou ukládány do fronty a odloženy
4. Provedení modifikujících požadavků
5. Provedení skriptu pro výpočet statistik
6. Vyhodnocení ukončovací podmínky – v případě že podmínka není splněna, vyčištění fronty modifikujících požadavků a skok na bod 3
7. Uchování statistik, přechod do stavu „Konec simulace“ - stav, v němž jsou klientům poskytovány pouze statistické informace

4.2.2 Algoritmus simulátoru – vedlejší proces

1. Čekání na připojení klienta
2. Jakmile dojde k připojení klienta, je spuštěn další proces čekání na klienta
3. Autorizace klienta a zaslání inicializačních dat klientovi
4. Provádění požadavků klienta a odesílání výsledků
5. Při přerušení spojení uvolnění prostředků pro režii komunikace s klientem

4.2.3 Algoritmus klienta vizualizace

1. Připojení k serveru a autorizace
2. Vystavění obrazu světa podle inicializačních dat
3. Vyslání požadavku
4. Zobrazení změn do pohledu na vizualizaci
5. Pokud není vizualizace přerušena, pokračovat bodem 3

4.3 Možnosti paralelizace

Tato kapitola popisuje možnosti paralelizace agentního systému takového, jaký byl v předchozí kapitole navržen. Metodologie zkoumání možností paralelizace je shora dolů – to znamená od globálního pohledu na celý systém směrem k jednotlivým podproblémům. Algoritmus a jeho reprezentace pomocí programu jsou definovány v základě jako posloupnost elementárních úkonů. Úkolem paralelizace je tedy najít takové úkony, které sebou nejsou navzájem ovlivněny – to znamená výsledky jednoho úkonu neovlivňují provedení druhého. Jistou výhodu při paralelizaci představuje povaha simulátoru agentního systému – simuluje totiž paralelní jevy. Simulaci lze rozdělit do tří etap.

4.3.1 Etapa inicializace

V této etapě se zpracovávají konfigurační soubory, vytváří se mapa, umisťují se na ni agenti a inicializují se atributy agentů. V této fázi by bylo možné paralelizovat načítání konfiguračních souborů, vzhledem k tomu, že jsou uspořádány do stromové struktury a dále by bylo možné paralelizovat provádění inicializace agentů. Tato etapa však nemá při typickém nasazení velký podíl na čase potřebném k simulaci.

4.3.2 Etapa provádění simulace

Tato etapa spočívá v opakovaném provádění skriptů chování každého z agentů. Akce agentů jsou už z principu paralelní – je ale třeba zajistit, aby jejich paralelní provádění nezpůsobilo chyby a nekonzistenci dat. Obecně je čtení operace bezpečná pro paralelní přístup a nebezpečí vzniká jen při zápisu. Veškeré akce, které provádí zápisové operace, se ale v navržené koncepci odkládají a zapisují se do fronty modifikujících požadavků. Toto řešení působí, že jedinou oblastí potenciálního konfliktu provádění skriptů chování agentů je právě fronta modifikujících požadavků.

Fronta modifikujících požadavků je další místo, kde by bylo možné nasadit paralelizaci. Zde je ovšem problém synchronizace přístupu k datům mnohem výraznější. Je třeba řešit problém priorit akcí, pokud to bude definováno, souběžný přístup do datových struktur agentů a mapy prostředí a problém zachování determinismu, kdy sekvenčně vyvolané požadavky na zápis musí vyústit ve výsledek shodný se sekvenčním provedením odpovídajících modifikujících akcí.

Další a zcela nezávislou oblastí je obsluha požadavků klientů vizualizace. Musí probíhat paralelně s výpočtem simulace. Každá obsluha musí být v samostatném vláknu už z principu, problém, který je zde potřeba řešit, je však synchronizace s výpočtem simulace. Tady je potřeba najít optimální způsob řízení přístupu k datům.

Vzhledem k tomu, že tato etapa je výpočetně nejnáročnější na celé simulaci, zaměřím se při paralelizaci primárně právě na ni. Tuto etapu lze dále rozdělit na fáze provádění skriptů agentů, provádění modifikujících akcí, výpočet statistik a vyhodnocení ukončovací podmínky. Výpočet statistik a vyhodnocení ukončovací podmínky jsou akce, které nelze obecně paralelizovat – paralelizace by musela být definována programátorem přímo v uživatelském skriptu. Tato možnost však nebyla v rámci návrhu zahrnuta. Zbývá tedy paralelizace procesů chování agentů a modifikující akce. Procesy agentů mohou být díky vyloučení modifikujících akcí prováděny zcela paralelně. Bude ale nutné zjistit, jak moc efektivní paralelizace bude. Bude potřeba prozkoumat situace, kdy budou procesy rozděleny do skupin a tyto pak budou prováděny paralelně, zatímco procesy uvnitř skupin budou prováděny sekvenčně. Bude také zajímavé prozkoumat případné přerozdělení procesů do skupin na základě experimentů za účelem optimalizace výpočtu – především jakým způsobem tyto skupiny procesů vytvářet a jak velké by skupiny procesů prováděné nezávislými paralelními vlákny měly být.

Pro paralelizaci provádění modifikujících akcí by bylo vhodné rozdělit tyto akce do skupin tak, aby pokud možno v jedné skupině byly akce týkající se konkrétních dat, aby při jejich provádění nebyla blokována jiná vlákna. Zajištění výše zmíněné podmínky však přináší značné zesložitění a po zběžné analýze pravděpodobně i nemalý nárůst spotřeby výpočetních zdrojů a paměti.

Je možné paralelizovat i samotné skripty agentů tak, aby výpočet chování jednoho agenta běžel ve více vláknech. Toto však vzhledem k povaze simulátoru pravděpodobně přinese užitek jen ve výji-

mečných případech. Problémem tohoto řešení je nutná podpora na straně vestavěného interpretu skriptu a nutná spolupráce programátora chování agenta, který musí zvládnout problematiku paralelizace.

4.3.3 Etapa konce simulace

Tato etapa už nepředstavuje velký problém – simulátor už jen odesílá statistiky na požádání klientům. Data už se nemění a proces čtení lze bez větších komplikací provádět paralelně. Tato etapa však nemá vliv na rychlosti ani efektivitě výpočtu simulace.

4.4 Paralelizace v návrhu

Po prozkoumání možností paralelizace, zhodnocení přínosu jednotlivých bodů a zvážení složitosti implementace, byla navržena konečná podoba jádra simulátoru z hlediska paralelizace. Etapa inicializace a etapa konce simulace nebyly paralelizovány vůbec – vzhledem k času potřebnému pro běh výpočtu simulace tyto etapy hrají méně významnou roli. Pozornost byla zaměřena na etapu provádění simulace.

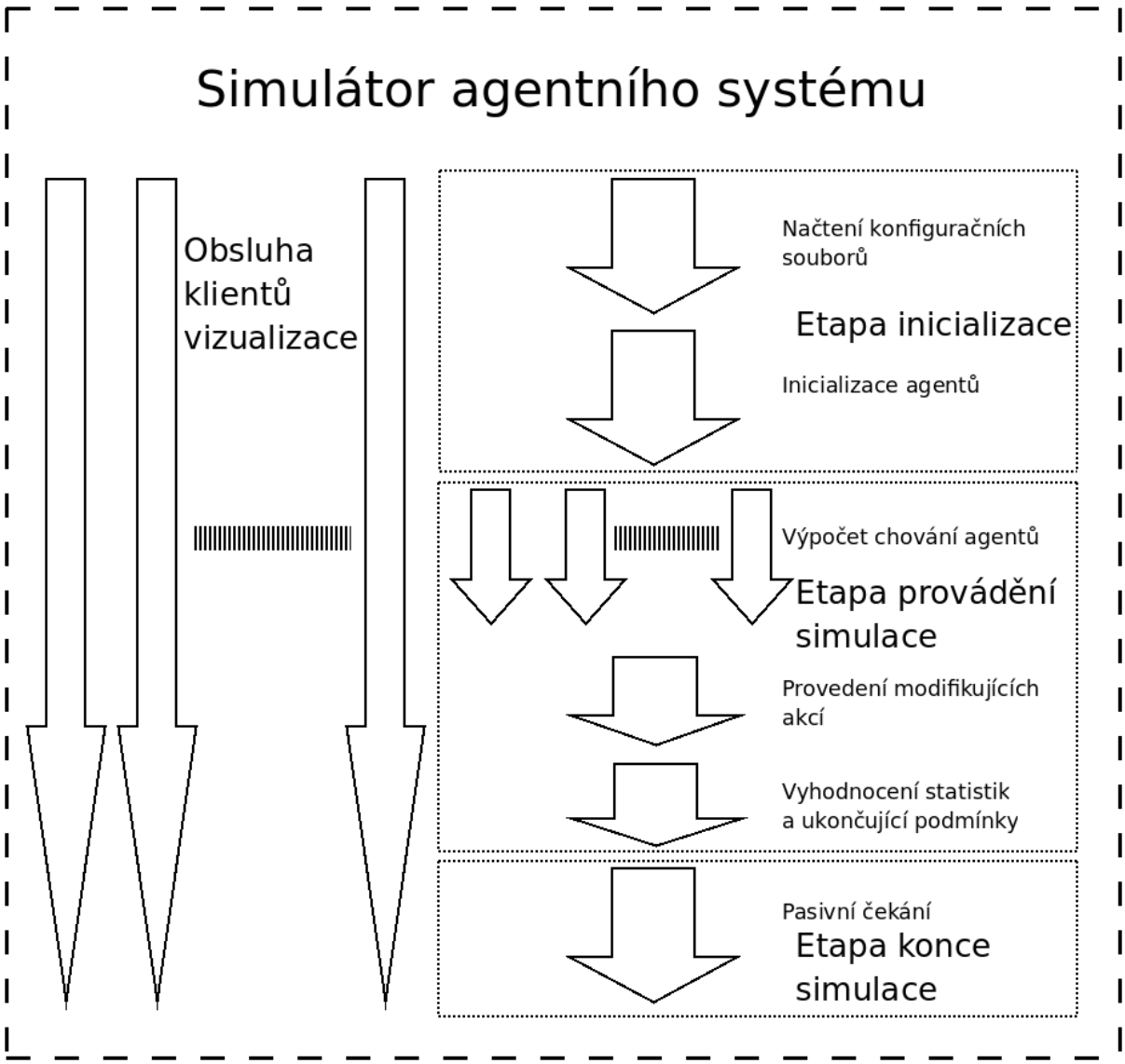
První výskyt paralelního zpracování nastává v oblasti komunikace s klienty vizualizace. Těch může být libovolně mnoho a potřebují číst stav simulátoru, který se během simulace mění. Tato situace byla vyřešena vláknovým zpracováním, kdy obsluha každého klienta vizualizace je spuštěna ve vláknu nezávislém na vláknu výpočtu simulace. Přístup k datům simulace během výpočtu procesů chování agentů je bez problémů možný a během provádění modifikujících akcí je výlučný přístup zajistitelný mutexem.

Nejvýznamnější oblastí paralelizace v současném návrhu je paralelizace výpočtu chování agentů. Potenciálním bodem kolize paralelního výpočtu chování agentů je fronta modifikujících požadavků – zde je nezbytné zajistit synchronizaci jednotlivých výpočetních vláken. Jinak ale může výpočet probíhat bez omezení paralelně. Díky této možnosti bylo možné navrhnout a otestovat více přístupů k paralelizaci. Přístup, který se nabízí jako první spočívá v rozdělení mapy na oblasti, které se zpracovávají nezávisle a současně. V návrhu je mapa dělena na obdélníky tak, aby každému vláknu připadala ideálně stejná plocha mapy. Vlákno vždy zpracovává jen ty agenty, kteří jsou na ploše mapy, jež mu byla přidělena. Tento přístup bude nadále označován jako *dělení mapy*. Dalším přístupem, který je možný díky tomu, že celý simulátor běží na jednom počítači se společnou pamětí, je dělení jednotlivých agentů do skupin. Tyto skupiny jsou pak přidělovány jednotlivým vláknům. V návrhu, kde jsou agenti identifikování svým jednoznačným ID, je dělení založeno právě na tomto identifikátoru. Toto řešení umožní lépe rovnoměrně rozdělit množinu agentů jednotlivým vláknům, která je zpracovávají.

Kromě přístupů k paralelnímu zpracování hraje roli i granularita tohoto dělení a případně které konkrétní podmnožiny agentů jsou zpracovávány kterými vlákny. Granularita patří k rozhodujícím faktorům efektivity paralelního zpracování. Příliš nízká granularita nevyužívá potenciál výpočetního systému, příliš vysoká pak přináší neefektivní režii. Použitá knihovna TBB umožňuje granularitu řídit automaticky. Pro účely porovnání je součástí návrhu i možnost ručně určit počet výpočetních vláken a tím i granularitu. Toto bylo navrženo proto, že cílem práce není jen simulátor jako takový, ale především průzkum a zhodnocení jednotlivých přístupů a způsobů zpracování simulace.

Paralelizace modifikujících akcí byla v tomto návrhu vypuštěna. Především pro složitost implementace, kde by bylo netriviální dosáhnou paralelního zpracování takového, aby režie paralelismu nepřevážila přínos tohoto zpracování, ale také proto, že při experimentech s návrhovými modely se ukázalo, že provedení modifikujících požadavků je ve srovnání s výpočtem chování agentů řádově méně časově náročné. Podobně paralelizace samotného skriptu agenta.

Konečná podoba paralelizace v navrženém simulátoru je zobrazena na obrázku 4.1.



Ilustrace 4.1: Schéma paralelismu v návrhu

4.5 Diskuze k návrhu

Návrh byl prováděn s cílem využít pokud možno existující technologie – jednak z důvodu zjednodušení návrhu, jednak proto, aby byly výsledky této práce snadno přenositelné a znovupoužitelné v jiných projektech.

Pro uchování vstupních a výstupních dat simulace byl zvolen jazyk XML. XML byl zvolen z toho důvodu, že je definován normou, nezávislý na softwarové platformě a procesorové architektuře a implementace knihoven pro práci s ním je běžně dostupná. V této práci konkrétně byla použita knihovna TinyXML [10].

Jazyk LUA byl zvolen coby jazyk pro skripty simulátoru. Tento jazyk byl zvolen pro svou úplnost – lze jím zapsat libovolný algoritmus. LUA je dynamicky typovaný – proměnné nemají typ, pouze hodnotu. Z řídicích konstrukcí jazyk LUA obsahuje konstrukce pro vytváření podmíněných příkazů, počítaných cyklů, podmíněných cyklů a bloků příkazů. Významnou vlastností je podpora funkcí, které bylo využito při vytváření API agenta a simulátoru. Zajímavou, ale z důvodu kompatibi-

lity v API simulátoru nepoužitou vlastností je možnost vracet libovolný počet hodnot z funkce. Bez vytváření datových struktur tak může funkce vracet například souřadnice bodu v prostoru. Této vlastnosti lze s výhodou využít při programování konkrétních simulací. Zásadním argumentem pro nasazení právě jazyka LUA je i dostupnost knihoven pro jeho interpretaci s rozhraním v jazyce C. V neposlední řadě byl brán zřetel na nezávislost tohoto jazyka – specifikace jazyka a implementace knihoven nepodléhá restriktivním licencím a není intelektuálním vlastnictvím konkrétního subjektu. Podrobné informace o tomto jazyce a jeho možnostech podává položka použité literatury Lua 5.1 Reference Manual [11]. Nevýhodou nasazení LUA pak může být jeho komplexita. A to jak z hlediska programátora, který musí zvládnout celý univerzální jazyk, tak i z hlediska interpretace, která je oproti jiným minimálním jednoúčelovým jazykům výpočetně náročnější. Navíc je nutno si uvědomit, že ne všechny možnosti jazyka LUA jsou v simulátoru využitelné. Například vestavěná funkce pro tisk hodnoty na standardní výstup se sice provede, ale výstup se ve vizualizaci neprojeví. Zmíněné problémy by pravděpodobně vyřešil specializovaný jazyk vyvinutý právě pro tuto práci. Toto řešení by však kromě zvýšení náročnosti implementace přineslo i jistá omezení v rozsahu a komplexitě implementovaných algoritmů v chování agenta a simulátoru. Takovým omezením může být například absence funkcí a podprogramů.

Jako formát ukládání obrazové informace byla zvolena znaková pseudografika. Pro vizualizaci agentních systémů je znakový mód přiměřený a dostačující ve většina případů. Další výhodou tohoto řešení je v porovnání především s rastrovou grafikou významně menší spotřeba paměti. Z hlediska simulace samotné nemají vizualizační data vliv – podpora jiných způsobů zobrazení a jiných formátů obrazových dat je v podstatě záležitostí klienta vizualizace.

Návrh je koncipován tak, aby bylo možné použít i případně jiné skriptovací jazyky nebo jiné formáty ukládání obrazových dat. Tyto však nemusí být podporovaný každou implementací simulátoru. Implementace v rámci mé práce je v tomto směru minimální.

Dalším předmětem diskuze je způsob provádění akcí agentů a především zpracování požadavků na zásahy do prostředí a stavu ostatních agentů. Použitá technika vytváření fronty modifikujících požadavků funguje tak, že při provádění procesů chování jednotlivých agentů se veškeré požadavky, které zasahují do mapy nebo atributů jiných agentů shromažďují na jednom místě – ve frontě modifikujících požadavků. Tato fronta se na konci kroku provede, jak bylo popsáno v popisu principu simulátoru. Rozšířením tohoto řešení by bylo použití tzv. „dvojitého bufferu“. Toto řešení by představovalo zdvojení veškerých dat potřebných pro běh simulace. Zatímco by se prováděly modifikující akce na jednom „bufferu“, druhý „buffer“ by sloužil pro čtení požadavků klientů. Toto řešení by umožnilo plynulejší obsluhu více vizualizačních klientů a nevyžadovalo by frontu modifikujících požadavků, jejíž provedení je závislé na dokončení akcí všech agentů, ale přineslo by téměř dvakrát vyšší nároky na paměť – proto nebylo v návrhu použito.

Posledním bodem je pak koncept klient-server. Toto řešení přináší zesložitění ovládání a nutnost spouštět více nezávislých aplikací v rámci jedné simulace. Navíc se většinou nepředpokládá, že by simulaci sledovalo velké množství uživatelů. Z tohoto pohledu by bylo vhodnější zabudovat vizualizaci přímo do simulátoru. Výhodou síťového řešení je ale skutečnost, že vizualizace nemusí běžet na stejném stroji jako samotná simulace. Vzhledem k výpočetní náročnosti některých simulací může nastat potřeba spustit simulaci na výkonném stroji – typicky výpočetním serveru, kde ale obvykle chybí možnost vizualizace. Obvyklým postupem pak bývá spuštění vzdálené vizualizace na klientském počítači. Ke vzdálené vizualizaci lze sice využít architekturu X nebo vzdálený terminál, ale toto řešení není v základě dostupné na všech používaných operačních systémech. Vizualizace po síti je v tomto ohledu dostupnější.

5 Implementace

Kapitola Implementace popisuje implementační detaily simulátoru. Je členěna do podkapitol, ve kterých popisuje třídy v rámci implementace, moduly, ze kterých je výsledný simulátor sestaven a v poslední kapitole podrobněji rozebírá implementaci jádra simulátoru. Tato kapitola se nevěnuje implementaci klienta vizualizace, protože tento je nezávislou částí a může být libovolně nahrazen. Navíc klient vizualizace nemá vliv na průběh simulace.

5.1 Popis tříd

Jak již z názvu kapitoly vyplývá, byla zvolena objektová implementace. Objekty totiž umožňují lépe organizovat funkční části tak, jak spolu souvisí a pro člověka je objektový přístup přehledný i při větším rozsahu projektu. Třídy byly vytvořeny tak, aby zahrnovaly vždy určitý tematický celek. Samostatnou třídu má např. agent, simulátor, nebo konfigurační soubor. Celý popis tříd je následující:

Simulation je třída představující jednotlivou instanci simulace. Tato třída je instanciována hlavní funkcí programu. Realizuje výpočet simulace. Poskytuje metody pro výpočet simulačního kroku, pro provedení modifikujících akcí, pro vyhodnocení ukončovací podmínky, pro získání aktuálních i statických dat simulace a metodu pro inicializaci z konfiguračního souboru. Právě metoda výpočtu simulačního kroku *Step()* je rozhodující součástí simulátoru – zde se odehrává řízení paralelismu a synchronizace vláken při výpočtu chování agentů. Metoda simulačního kroku v třídě *Simulation* při běžném provozu simulátoru představuje část programu, kde procesor tráví nejvíce času.

Agent je třída představující agenta. Každý agent v simulaci je reprezentován jednou instancí třídy *Agent*. Třída poskytuje metody pro vytvoření agenta (konstruktor), nastavení pozice agenta, čtení a nastavování atributů agenta a především metodu pro provedení hlavního skriptu agenta *RunMainScript(int SIM_ID, int thread_ID)*. Metodou metatřídy třídy *Agent* je potom *InitAgentType(std::string name, std::string agentXML)*, která realizuje načtení profilu agentů z konfiguračního souboru. Jméno profilu *name* je pak použito při vytváření agentů, kterým jsou pak přiřazeny data a vlastnosti profilu.

Map je třída realizující prostředí, v němž simulace probíhá – mapu. Každá simulace reprezentovaná instancí třídy *Simulation* instanciuje jedenkrát třídu map. Instanciace mapy vyžaduje cestu ke konfiguračnímu souboru mapy, podle kterého je mapa sestavena. Třída *Map* poskytuje kromě jiných metody pro vytvoření mapy, přidání a odstranění agenta, změnu pole mapy a získání seznamu agentů na poli mapy. Mapa je pravoúhlá. Jádrem mapy je asociativní pole, které mapuje souřadnice na odpovídající strukturu pole mapy.

SimConfig je třída reprezentující parser konfiguračního souboru simulace. Při inicializaci simulace je tento konfigurační soubor načten a zpracován a získaná data vložena do simulace. *SimConfig* poskytuje metody pro získání dat jako jsou maximální počet kroků simulace, skript pro výpočet statistik, skript pro výpočet ukončovací podmínky, cestu ke konfiguračnímu souboru mapy, seznam typů agentů a odpovídající odkazy na konfigurační soubory a jiné. Konfigurační soubor je zpracován při instanciaci třídy a po dobu její existence jsou odpovídající data dostupná.

MapConfig je třída představující parser konfiguračního souboru mapy. Podobně jako předchozí je třída instanciována při inicializaci mapy. Při instanciaci zpracuje konfigurační soubor předaný konstruktoru a uloží do paměti požadovaná data. Metodami třídy lze tato data získat. Třída poskytuje metody pro získání rozměrů mapy a vzorku mapy zadaného v konfiguračním souboru.

AgentConfig je třída reprezentující parser konfiguračního souboru agenta. Instanciuje se při volání metody *InitAgentType(std::string name, std::string agentXML)* metatřídy třídy *Agent*. Při instanciaci *AgentConfig* zpracovává konfigurační soubor a ukládá data do paměti, kde jsou dostupná pomocí metod třídy. Třída poskytuje mimo jiných metody pro získání inicializačního skriptu agenta a hlavního skriptu agenta. Tyto skripty jsou výsledkem konverze různých zápisů inicializace agenta na výsledný skript. Definované vnitřní stavy agenta jsou při převodu konvertovány na odpovídající funkce z rozhraní agenta. Ačkoliv pro metadata třídy *Agent* je rozhodující asociace typ agenta – odpo-

vídající data, třída *AgentConfig* žádné informace o typu agenta nenese. Typ je k datům přiřazen až samotnou metodou *InitAgentType(std::string name, std::string agentXML).*

Arg je třídou reprezentující rozhraní parametrů z příkazové řádky. Při instanciaci přebírá a zpracovává argumenty *argc* a *argv* – standardní argumenty reprezentující parametry z příkazové řádky a zpracovává je. Pomocí metod třídy pak lze ověřit existenci jistého parametru, získat data následující za parametrem a získat aktuální cestu.

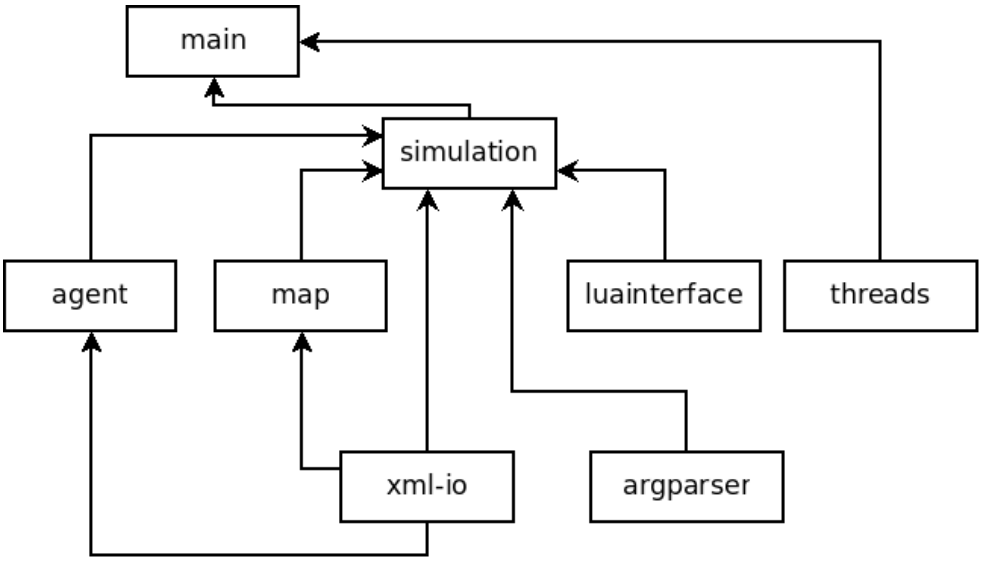
V tomto popisu tříd nebyly popsány všechny metody. Vynechávány byly především metody související s vizualizací a získáváním dat pro ni, jednak proto, že je jich poměrně hodně a popis by protáhly a znepřehlednily a také proto, že přímo nesouvisí se simulací. Podrobnější popis toho, jak jsou třídy implementovány, jak jsou členěny do modulů a jakých algoritmů a datových struktur je v jejich implementaci využito je v následujících kapitolách.

5.2 Popis modulů

Implementace je rozdělena na moduly podle jednotlivých logických celků. Moduly implementují buď jednu rozsáhlou třídu, více menších souvisejících tříd, nebo sadu funkcí související s konkrétní problematikou. Detaily o každém modulu jsou uvedeny v následujícím popisu.

- Modul *main* je modul implementující hlavní funkci programu. Obsahuje vlákno obsluhující simulaci a vlákno pro obsluhu síťové komunikace. V tomto modulu je také kód zajišťující bezpečný přístup k simulačním datům pro síťového klienta.
- Modul *simulation* je nejvýznamnější a nejrozsáhlejší modul aplikace. Implementuje třídu *Simulation* a její metody. Tento modul obsahuje kód paralelizace kroku simulace.
- Modul *agent* implementuje třídu *Agent*. Tento modul neobsahuje žádný kód týkající se paralelizace výpočtu. Uchovává pouze kód jednotlivých skriptů a asociativní pole stavů agenta. Z použitých knihoven závisí pouze na LUA, neboť obsahuje strukturu LUA zásobníku, které se využívá při interpretaci skriptů.
- Modul *map* je modul implementující mapu nad kterou probíhá simulace – třídu *Map*. Stejně jako modul *agent*, ani *map* neobsahuje žádný kód související s paralelizací výpočtu. To je možné díky algoritmu s frontou modifikujících požadavků.
- Modul *luainterface* implementuje sadu funkcí potřebných pro interpretaci LUA skriptů při simulaci. Obsahuje LUA rozhraní jak agenta, tak simulace.
- Modul *xml-io* implementuje třídy týkající se práce s XML konfiguračními soubory. Jedná se o třídy *SimConfig*, *MapConfig* a *AgentConfig*. Pro zpracování XML je modul velmi úzce spjat s knihovnou TinyXML.
- Modul *threads* je modul obsahující na platformě nezávislou implementaci vláken. Využívá ho modul *main*. Od verze 2.1 knihovny TBB už jsou vlákna obsažena přímo v knihovně. Vývoj ale začal na starší verzi a z důvodu kompatibility byl modul zachován. V současnosti už ale není nezbytný a může být snadno nahrazen.
- Posledním modulem je modul *argparser*. Tento modul slouží ke zpracování parametrů z příkazové řádky. Implementuje třídu *Arg.*

Na obrázku 5.1 je zobrazena hierarchie modulů. Šipkami jsou vyznačeny závislosti modulů – směr šipky je od využívaného modulu k závislému.



Ilustrace 5.1: Diagram modulů

5.3 Implementace jádra simulátoru

Jádro simulátoru je v tomto případě nejdůležitější částí aplikace – proto je mu věnována samostatná podkapitola. Jádrem simulace jsou zde myšleny metody a funkce pro výpočet kroku simulace a vyhodnocení ukončovacích podmínek. Jádro simulace je implementováno v třídě *Simulation*.

Samotný průběh simulace musí být řízen – je třeba provádět kroky simulace a hlídat stav ukončovací podmínky – toto řeší jak už bylo výše zmíněno hlavní vlákno aplikace v modulu *main*. Cyklické provádění kroků je zajištěno jednoduchým podmíněným cyklem, který se opakuje, dokud nenastane kladné vyhodnocení ukončovací podmínky. To může nastat buď vyhodnocením skriptu podmínky, nebo dosažením maximálního počtu kroků zadaného v konfiguračním souboru. Ve výchozím nastavení po dokončení simulace vlákno obsluhy simulace končí a zůstávají běžet pouze vlákna obsluhy vizualizačních klientů. Za běhu přistupují vlákna vizualizačních klientů k datům simulace souběžně s vláknem řídícím simulaci prostřednictvím výpočtu simulace. Pro výlučný přístup je použit read-write mutex z knihovny TBB.

Nyní rozebereme podrobněji samotný krok simulace. Nejdříve, jak už bylo popsáno výše, se provede výpočet chování agentů – interpretují se jejich hlavní skripty. Toto je kritická fáze simulace a právě zde je jádro celé práce. Obecně je nutné projít všechny agenty a provést jejich skripty. Postupně od nejnižšího ID po nejvyšší to provádí jedno-vláknové zpracování. Postupně projde asociativní pole agentů, které ukládá instance třídy Agent na základě ID a u každé instance zavolá metodu *RunMainScript* (viz. Kapitola 5.1). Metoda provede skript agenta a akce, které by měly modifikovat stav simulace, skládá do fronty modifikujících požadavků. U paralelního přístupu je právě provádění akcí agentů částečně souběžné ve více vláknech. Zde se liší přístupy algoritmů dělení agentů a dělení mapy. Přístup dělení agentů dělí agenty na základě ID – pomocí prostředků knihovny TBB rozdělí množinu agentů podle ID na stejné úseky a ty potom předá ke zpracování jednotlivým vláknům. Dělení je buďto pevné tak, aby byl zachován určitý počet vláken, nebo automatické na základě partiotioneru (viz. kapitola 3.1.2). Každé vlákno pak sekvenčně projde podmnožinu agentů v rozsahu ID, které dostane přiděleno. Algoritmus dělení mapy postupuje zcela jinak. Nejdříve při inicializaci simulace je mapa rozdělena na množství submap. Jejich počet je zadán konstantou uvnitř aplikace. Při provádění skriptů agentů je množina submap rozdělena na podmnožiny a ty jsou přiřazeny jednotlivým výpočetním vláknům. Každá submapa má svůj index a právě tyto indexy jsou podobně jako u předchozího přístupu ID agentů přidělovány v určitém rozsahu každému výpočetnímu vláknu. Každé vlákno pak využitím metody třídy *Map* získá seznam agentů (instancí třídy *Agent*) na těchto submapách a po-

stupně u nich zavolá metodu *RunMainScript*. Tím vlákno zpracuje agenty na přidělené ploše mapy. Granularitu v obou algoritmech představuje velikost podmnožin. Dělení mapy na submapy se může v tomto případě zdát jako zbytečná komplikace. Má ale za cíl zajistit, aby vlákna dostávala přidělenou plochu vhodnějšího tvaru (ne třeba dlouhý pruh o šířce jednoho pole) a tím minimalizovat potřebu přistupovat k datům agentů zpracovávaných jinými vlákny. To má negativní vliv na efektivitu cachování dat.

Vytváření a synchronizaci vláken zajišťuje knihovna TBB. Data simulace jsou skripty agentů pouze čtena, takže zde problém souběžného přístupu nenastává. Jediným bodem potenciálního konfliktu je tedy fronta modifikujících požadavků, jak již bylo zmíněno v kapitole 4.4. Zde byla nasazena třída *concurent_queue* z knihovny TBB. Realizuje abstrakci nad frontou a zajišťuje bezpečný souběžný přístup k prvkům fronty. Stačí korektně používat metody a operátory třídy. Požadavky do fronty modifikujících požadavků jsou přidávány z vláken náhodně tak, jak se objevují. Jako optimalizace bylo použito řešení, kdy si každé vlákno vytváří svou lokální frontu, kterou ukládá do pole front. Tím se souběžný přístup a vzájemná blokace vláken zmenšuje na minimum. Ačkoliv je v celé práci zmiňována vždy jedna fronta modifikujících požadavků, ve skutečnosti je front více – jednotlivé fronty představují prioritu operací a operace jsou do nich podle své povahy přidávány.

Po provedení všech skriptů agentů následuje vykonání modifikujících požadavků. To se provádí už sekvenčně v jednom vlákně. Od nejnižší priority po nejvyšší se postupně projdou fronty modifikujících požadavků a všechny požadavky se zanesou do stavu simulace. V době provádění fronty modifikujících požadavků je přístup k datům simulace blokován. Především se to týká klientů vizualizace, kteří v této fázi nemohou získávat data. V případě, že se sejdou dva konfliktní modifikující požadavky (např. nastav atribut agenta na 10 a druhý nastav atribut agenta na 0), je proveden ten z nich, který byl vložen později.

Po dokončení všech změn stavu simulátoru je vyhodnocena ukončovací podmínka. Je proveden skript výpočtu statistik a po něm skript ukončovací podmínky. Po vyhodnocení podmínky je porovnán počet kroků s maximálním počtem kroků. Pokud je dosaženo limitu kroků, je vrácena kladná hodnota ukončovací podmínky nezávisle na tom, jakou hodnotu skript podmínky vrátil. To vede k ukončení vlákna řídícího simulaci. Po skončení vlákna simulace zůstávají všechny třídy související se simulací instanciovány. Data jsou tak uchována pro klienty vizualizace.

6 Testování

Pro ověření vlastností implementovaného simulátoru byla sestavena sada testů. Testy se zaměřují především na mezní využití simulátoru, ale obsahují i test reálnou simulací. Testování proběhlo na několika počítačích s různým počtem procesorových jader, aby bylo možno hodnotit přínos paralelismu s rostoucím počtem fyzických paralelních vláken. Před samotným testováním byly provedeny odhady předpokládaných výsledků některých testů. Tyto odhady jsou níže konfrontovány s reálnými výsledky. U testů jsou uvedeny pouze výsledné hodnoty. Celkové zhodnocení výsledků a vyvození závěrů je přenecháno do závěrečné kapitoly.

6.1 Návrh testů

6.1.1 Test systematického pohybu agentů

Cíl testu: Ověření chování simulátoru při pohybu velkého množství agentů v jednom místě mapy.

Inicializace agenta:
- Umístění agenta na pozici 39,39 (rozměr mapy je v obou směrech 0..39).

Skript agenta:
- Zjistí svou pozici.
- Změní terén mapy na poli, kde se nachází (přepíše symbol pole mapy na *'x'*). Tím vyznačí cestu.
- Vypočítá novou pozici – souřadnice X a Y sníží obě o 1.
- Přesune se na novou pozici.

Ukončení: 15 kroků simulace, 10000 agentů

Měřený parametr: Čas [s]

6.1.2 Test náhodného pohybu agentů

Cíl testu: Ověření chování simulátoru při náhodném pohybu velkého množství rovnoměrně rozmístěných agentů.

Inicializace agenta:
- Umístění agenta na náhodnou pozici.

Skript agenta:
- Zjistí svou pozici.
- Zvolí náhodně změnu této pozice v rozsahu jednoho pole (i úhlopříčně).
- Změní terén mapy na poli, kde se nachází (přepíše symbol pole mapy na *'x'*). Tím vyznačí cestu.
- Přesune se na novou pozici.

Ukončení: 15 kroků simulace, 10000 agentů

Měřený parametr: Čas [s]

6.1.3 Test reálnou simulací

Cíl testu: Ověření chování simulátoru v reálném nasazení.

Inicializace agenta:

- Inicializace atributu agenta *life* na hodnotu 10.
- Umístění agenta na náhodnou pozici.

Skript agenta:

Agent typu 0:

- Pokud zjistí hodnotu svého atributu *life* menší nebo rovnu 0, odstraní sám sebe.
- Prohledá všechna pole v okolí jednoho pole kolem sebe (i úhlopříčně).
- Pokud narazí na agenta cizího typu, zaútočí (sníží hodnotu jeho atributu *life* o 1).
- Pokud neprovedl útok, přesune se náhodně o jedno pole vertikálně nebo horizontálně, nebo zůstane na místě.

Agent typu 1:

- Pokud zjistí hodnotu svého atributu *life* menší nebo rovnu 0, odstraní sám sebe.
- Prohledá všechna pole v okolí 4 polí kolem sebe (i úhlopříčně) a sečte všechny agenty svého typu a všechny cizího typu.
- Najde nejbližšího agenta cizího typu.
- Pokud je počet agentů stejného typu větší nebo roven počtu agentů cizího typu a pokud je v těsné blízkosti (v rozsahu 1 pole) agenta cizího typu, zaútočí (sníží hodnotu jeho atributu *life* o 1).
- Pokud je počet agentů stejného typu větší nebo roven počtu agentů cizího typu a pokud není v těsné blízkosti (v rozsahu 1 pole) agenta cizího typu, zvolí další krok pohybu směrem k nejbližšímu cizímu agentovi.
- Pokud nezaútočil a ani nenastavil směr pohybu k nejbližšímu agentovi, zvolí směr pohybu náhodně (může zůstat i stát).
- Přesune se ve zvoleném směru.

Kód je zapsán v jazyce LUA – ukázka:

```
--[[ pokud nema zivot, spacha 'sebevrazdu' --]]
if (0 >= GetAttrib('life')) then
  Destroy();
else
  ...
end;
```

Ukončení: 40 kroků simulace, 2000 agentů

Měřený parametr: Čas [s]

6.2 Testovací výpočetní systémy

Číslo	Typ CPU	Počet CPU/jader	Architektura	Operační systém
1	PIII (Coppermine) @ 800MHz	1	x86	Linux 2.6.24-20-generic
2	Core2Duo T7500 @ 2.20GHz	2	x86_64	Linux 2.6.27-11-generic
3	Dual-Core Opteron 2220 @ 2.80GHz	4	x86_64	Linux 2.6.18-6-amd64
4	Xeon X5355 @ 2.66GHz	8	x86_64	Linux 2.6.18-6-amd64

Tabulka 1: Testovací výpočetní systémy

6.3 Předpokládané výsledky

Předpokládané výsledky jsou odhady založené na známých faktech a logických úvahách. Nejedná se o výsledky, které musí nutně testování potvrdit. Předběžně lze vyslovit obecné předpoklady, které splňují i mnohé jiné paralelní projekty. Především lze předpokládat, že paralelní zpracování přinese nárůst výkonu, pokud bude mít výpočetní systém více než jedno procesorové jádro a bude-li se jednat o SMP (Symetric Multi-Processor), což jsou všechny více-jádrové systémy, na nichž bylo testování prováděno. U takových systémů jsou všechna výpočetní jádra rovnocenná co do možností i výkonu. Dále je možné odhadnout, že na systémech s více procesory bude efektivnější využití většího počtu vláken. Stejně tak je pravděpodobné, že u systému s jedním procesorem nepřinese paralelismus výrazný přínos. Dokonce je možné, že vícevláknové zpracování bude na tomto systému pomalejší, než zpracování jedno-vláknové. Níže v této kapitole pokračují konkrétní předpoklady pro jednotlivé testy.

6.3.1 Test systematického pohybu agentů

Tento test intenzivně zatěžuje vždy dvě buňky mapy – jednu, kterou agenti opouští a druhou, na kterou vchází. U přístupu dělení agentů lze předpokládat zrychlení odpovídající počtu vláken prováděných souběžně, avšak u přístupu dělení mapy toto zrychlení předpokládat nelze, neboť se všechny skripty agentů budou zpracovávat v rámci jednoho segmentu mapy a tudíž i jednoho vlákna. Oproti jednovláknovému přístupu lze dokonce předpokládat i mírné zpomalení, neboť režie vytváření a synchronizace vláken se přičítá k celkovému času potřebnému pro výpočet simulace.

6.3.2 Test náhodného pohybu agentů

U tohoto testu je předpoklad, že přístup dělení agentů a přístup dělení mapy přinesou přibližně stejné výsledky, protože agenti jsou rovnoměrně rozprostřeni po mapě a tudíž při dělení mapy jsou jednotlivá výpočetní vlákna rovnoměrně zatěžována.

6.3.3 Test reálnou simulací

V tomto případě nelze jednoduše předpovědět, jaké budou výsledky testů. Testovány jsou současně různé subsystémy simulátoru. Přesto lze však vyslovit několik předpokladů. Kromě zrychlení výpočtu při nasazení více vláken jak už bylo zmíněno výše lze očekávat, že přístup dělení agentů bude efektivnější, než přístup dělení mapy. Je to proto, že při dělení agentů lze lépe zajistit přiměřeně rovnoměrné rozložení agentů na jednotlivá vlákna výpočtu. U dělení mapy toto rozložení závisí na poloze agentů na mapě a ti nemusí být mnohdy výhodně rozloženi.

6.4 Výsledky testů a konfrontace s předpoklady

V této kapitole jsou popsány výsledky testů. Výsledky jsou zapsány ve formě tabulky, kde kromě absolutní časové hodnoty v sekundách je uvedeno i procentuální relativní zrychlení výpočtu proti referenčnímu jednovláknovému řešení a procentuální relativní zrychlení pouze paralelizovatelné části. Měření probíhala utilitou *time*. Jako směrodatný parametr byl využit reálný čas výpočtu (položka real). Byla snaha provádět všechny testy na konkrétním výpočetním systému za stejných podmínek. Za účelem snížení chyby byl každý test spuštěn opakovaně a výsledky byly zprůměrovány. Takto byla získána data, která jsou nyní zobrazena v tabulkách. Procentuální relativní zrychlení je potom vypočítáno vzorcem (1).

$$Z_{rel} = \frac{t_{ref} - t_{\mathrm{sim}}}{t_{ref}} \cdot 100 \qquad (1)$$

Kde: Z_{rel} ... procentuální relativní zrychlení [%]
t_{ref} ... referenční čas jedno-vláknového zpracování [s]
t_{sim} ... čas výpočtu simulačního testu [s]

Na základě Amdahlova zákona (podrobnosti viz. [12]) je určena paralelizovatelná část systému. Amdahlův zákon definuje vztah (2).

$$T_{rel} = 100 - P + \frac{P}{n} \qquad (2)$$

Kde: T_{rel} ... relativní část času výpočtu vzhledem k času jedno-vláknového zpracování [%]
P ... paralelizovatelná část systému [%]
n ... počet výpočetních jader systému (předpokládá se SMP)

S využitím tohoto stavu je možno vytvořit vzorec pro výpočet času, který simulátor strávil pouze v paralelizovatelné části. Vztah (3) jej popisuje.

$$t_{simp} = t_{\mathrm{sim}} \cdot \frac{\left(\frac{n \cdot 100 - T_{rel}}{n-1} \right)}{100} \qquad (3)$$

Kde: t_{simp} ... čas výpočtu simulace strávený pouze v paralelizovatelné části aplikace
t_{sim} ... čas výpočtu simulace
n ... počet výpočetních jader systému (předpokládá se SMP)
T_{rel} ... relativní část času výpočtu vzhledem k času jedno-vláknového zpracování [%]

Procentuální relativní zrychlení samotné paralelizovatelné části je počítáno vzorcem č.1 dosazením času t_{simp}. Velikost paralelizovatelné části byla tam, kde to bylo možné, určena z dvou-vláknového zpracování algoritmem dělení agentů. Toto řešení bylo zvoleno proto, že s výjimkou výpočetního systému č.1 jsou na všech systémech dostupné dvě hardwarová vlákna a algoritmus dělení agentů není tak výrazně postižen specifičností testů – podává ve všech testech přiměřené výsledky. Na systému

č.1 nebylo možné Amdahlovým zákonem velikost paralelizovatelné části určit, neboť systém nedisponuje více než jedním výpočetním jádrem a zrychlení paralelizací nenastává.

Na závěr každého testu jsou konfrontovány výsledky testování s dříve vyslovenými předpoklady. Konfrontace jsou doplněny grafy závislosti počtu výpočetních vláken na čase zpracování. V grafech je přerušovanou čarou naznačen trend mezi jednotlivými výsledky.

Cílem testů není srovnání absolutních časů výpočtu mezi jednotlivými systémy, neboť tyto mají někdy i řádově rozdílné výkonnostní parametry, ale porovnání přínosu paralelizace na daném výpočetních systému vůči jednovláknovému referenčnímu zpracování.

6.4.1 Test systematického pohybu agentů

Testovací výpočetní systém č.1 PIII (Coppermine)@ 800MHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	420,32	420,96	422,05	423,00	422,35
Relativní zrychlení [%]	0,00	-0,15	-0,41	-0,64	-0,48

Tabulka 2: Výsledky testu systematického pohybu agentů – algoritmus dělení agentů na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	420,32	423,76	426,28	422,11	422,99
Relativní zrychlení [%]	0,00	-0,82	-1,42	-0,43	-0,63

Tabulka 3: Výsledky testu systematického pohybu agentů – algoritmus dělení mapy na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Testovací výpočetní systém č.2 Core2Duo T7500@ 2.20GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	87,70	52,18	52,26	52,40	52,14
Relativní zrychlení [%]	0,00	40,51	40,41	40,25	40,55
Relativní zrychlení [%] Paralelizovatelná část	0,00	49,99	49,88	49,68	50,05

Tabulka 4: Výsledky testu systematického pohybu agentů – algoritmus dělení agentů na výpočetním systému č.2.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	87,70	98,83	96,59	97,25	99,27
Relativní zrychlení [%]	0,00	-12,69	-10,14	-10,89	-13,19
Relativní zrychlení [%] Paralelizovatelná část	0,00	-15,66	-12,51	-13,44	-16,28

Tabulka 5: Výsledky testu systematického pohybu agentů – algoritmus dělení mapy na výpočetním systému č.2.

Testovací výpočetní systém č.3 Dual-Core Opteron 2220@ 2.80GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	74,63	43,99	26,48	27,43	27,45
Relativní zrychlení [%]	0,00	41,06	64,53	63,25	63,22
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,89	72,12	70,70	70,67

Tabulka 6: Výsledky testu systematického pohybu agentů – algoritmus dělení agentů na výpočetním systému č.3.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	74,63	76,04	76,19	75,16	74,55
Relativní zrychlení [%]	0,00	-1,88	-2,09	-0,70	0,11
Relativní zrychlení [%] Paralelizovatelná část	0,00	-2,11	-2,34	-0,79	0,12

Tabulka 7: Výsledky testu systematického pohybu agentů – algoritmus dělení mapy na výpočetním systému č.3.

Testovací výpočetní systém č.4 Xeon X5355@ 2.66GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	82,32	49,26	30,32	25,76	20,97
Relativní zrychlení [%]	0,00	40,16	62,17	68,71	74,53
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,52	71,16	77,85	84,47

Tabulka 8: Výsledky testu systematického pohybu agentů – algoritmus dělení agentů na výpočetním systému č.4.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	82,32	83,14	83,18	82,96	82,85
Relativní zrychlení [%]	0,00	-1,00	-1,04	-0,77	-0,64
Relativní zrychlení [%] Paralelizovatelná část	0,00	-1,13	-1,18	-0,88	-0,73

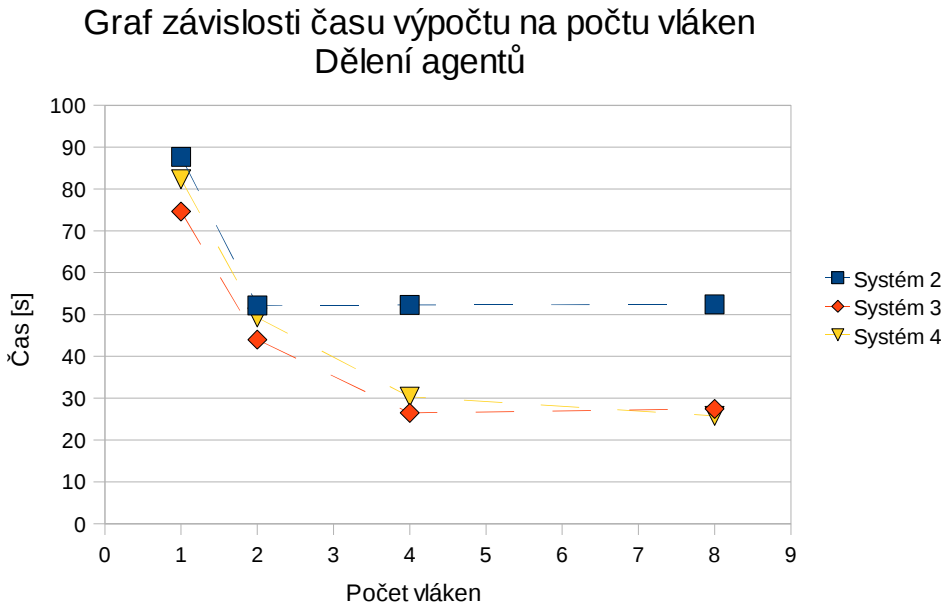
Tabulka 9: Výsledky testu systematického pohybu agentů – algoritmus dělení mapy na výpočetním systému č.4.

Konfrontace s předpoklady

Obecné předpoklady v případě prvního testu byly víceméně splněny. S rostoucím počtem vláken se stává výpočet efektivnější na více-jádrových systémech v závislosti na počtu jader. Tuto skutečnost vizualizuje graf č.1. Algoritmus dělení mapy toto pravidlo sice porušuje, avšak toto bylo předpokládáno vzhledem k povaze testu.

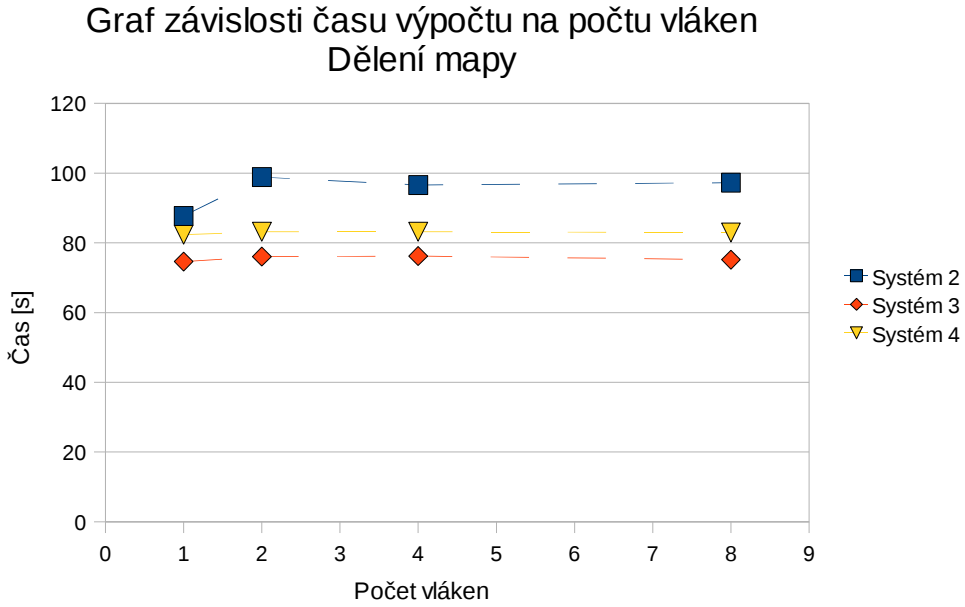
Bylo odhadnuto, že zatímco algoritmus dělení agentů přinese nárůst výkonu, tak algoritmus dělení mapy tento nárůst nepřinese. Tento předpoklad byl splněn. Zatímco v případě dělení agentů dochází k nárůstu výkonu až o necelých sedmdesát pět procent v případně testovacího systému č.4

s osmi výpočetními vlákny, v případě algoritmu dělení mapy dochází k mírnému výkonnostnímu propadu. Výjimku zde tvoří testovací systém č.2, kde je výkonností propad více než deset procent.



Graf 1: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení agentů

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.



Graf 2: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení agentů

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.

6.4.2 Test náhodného pohybu agentů

Testovací výpočetní systém č.1 PIII (Coppermine)@ 800MHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	427,62	429,33	430,53	427,62	430,72
Relativní zrychlení [%]	0,00	-0,40	-0,68	-0,37	-0,72

Tabulka 10: Výsledky testu náhodného pohybu agentů – algoritmus dělení agentů na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	427,62	430,04	429,68	429,67	427,84
Relativní zrychlení [%]	0,00	-0,57	-0,48	-0,48	0,00

Tabulka 11: Výsledky testu náhodného pohybu agentů – algoritmus dělení mapy na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Testovací výpočetní systém č.2 Core2Duo T7500@ 2.20GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	89,28	52,04	52,27	51,59	52,18
Relativní zrychlení [%]	0,00	41,73	41,45	42,22	41,56
Relativní zrychlení [%] Paralelizovatelná část	0,00	46,16	45,88	46,72	45,99

Tabulka 12: Výsledky testu náhodného pohybu agentů – algoritmus dělení agentů na výpočetním systému č.2.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	89,28	55,85	55,73	55,23	55,84
Relativní zrychlení [%]	0,00	37,45	37,58	38,14	37,91
Relativní zrychlení [%] Paralelizovatelná část	0,00	41,44	41,59	42,21	41,45

Tabulka 13: Výsledky testu náhodného pohybu agentů – algoritmus dělení mapy na výpočetním systému č.2.

Testovací výpočetní systém č.3 Dual-Core Opteron 2220@ 2.80GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	75,34	44,79	27,87	25,97	27,06
Relativní zrychlení [%]	0,00	40,55	63,01	65,53	64,09
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,68	70,98	73,83	72,20

Tabulka 14: Výsledky testu náhodného pohybu agentů – algoritmus dělení agentů na výpočetním systému č.3.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	75,34	47,41	29,24	27,56	26,04
Relativní zrychlení [%]	0,00	37,07	61,19	63,43	65,44
Relativní zrychlení [%] Paralelizovatelná část	0,00	41,77	68,94	71,45	53,72

Tabulka 15: Výsledky testu náhodného pohybu agentů – algoritmus dělení mapy na výpočetním systému č.3.

Testovací výpočetní systém č.4 Xeon X5355@ 2.66GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	83,47	49,78	29,88	24,14	21,06
Relativní zrychlení [%]	0,00	40,35	64,20	71,08	74,77
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,61	72,55	80,33	84,49

Tabulka 16: Výsledky testu náhodného pohybu agentů – algoritmus dělení agentů na výpočetním systému č.4.

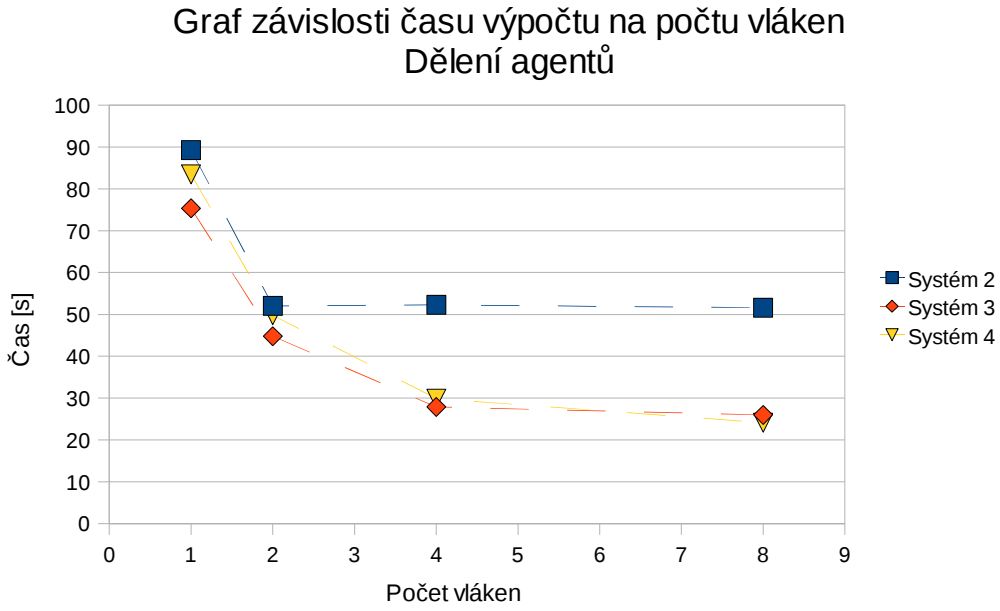
Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	83,47	52,59	32,76	28,51	24,57
Relativní zrychlení [%]	0,00	37,00	60,86	65,85	70,56
Relativní zrychlení [%] Paralelizovatelná část	0,00	41,81	68,65	74,41	79,74

Tabulka 17: Výsledky testu náhodného pohybu agentů – algoritmus dělení mapy na výpočetním systému č.4.

Konfrontace s předpoklady

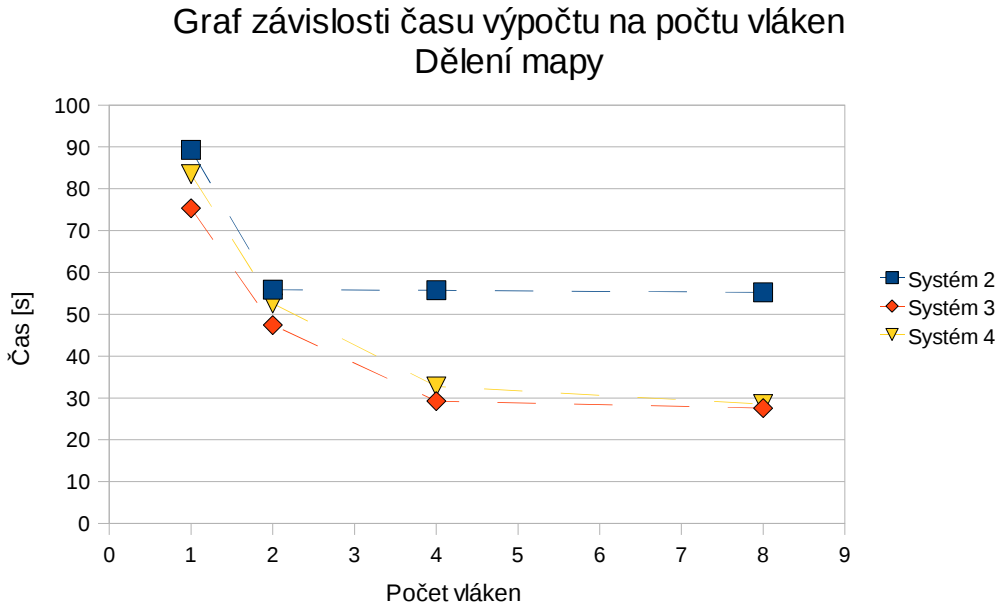
U tohoto testu byly obecné předpoklady splněny v celém rozsahu. Jedno-jádrový výpočetní systém dosáhl při testech s více vlákny v průměru malého zpomalení oproti jednovláknovému řešení. Naopak systémy s více jádry a procesory dosáhly u paralelního zpracování lepších výsledků, než u jedno-vláknového. Nejefektivnější počet vláken se prakticky zastavil na počtu procesorových jader systému – tedy i předpoklad, že u více-jádrových systémů bude výpočet ve více vláknech efektivnější byl v tomto testu bez výjimky splněn. Podrobnější vizuální srovnání představují následující grafy.

Konkrétní předpoklady týkající se tohoto testu byly taktéž splněny. Jak přístup dělení agentů, tak přístup dělení mapy přinesl nárůst výkonu. Zisk výkonu byl u přístupu dělení agentů většinou v řádu procenta až několika procent vyšší.



Graf 3: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení agentů

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.



Graf 4: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení mapy

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.

6.4.3 Test reálnou simulací

Testovací výpočetní systém č.1 PIII (Coppermine)@ 800MHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	138,34	141,40	138,77	141,78	139,09
Relativní zrychlení [%]	0,00	-2,21	-0,31	-2,48	-0,55

Tabulka 18: Výsledky testu reálnou simulací – algoritmus dělení agentů na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	138,34	139,71	139,82	139,48	139,87
Relativní zrychlení [%]	0,00	-0,99	-1,07	-0,82	-1,11

Tabulka 19: Výsledky testu reálnou simulací – algoritmus dělení mapy na výpočetním systému č.1. Zrychlení pouze paralelizovatelné části není obsaženo – nebylo jej možno určit.

Testovací výpočetní systém č.2 Core2Duo T7500@ 2.20GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	27,48	16,65	16,77	17,60	16,45
Relativní zrychlení [%]	0,00	39,43	38,99	35,94	40,16
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,20	44,70	41,24	46,03

Tabulka 20: Výsledky testu reálnou simulací – algoritmus dělení agentů na výpočetním systému č.2.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	27,48	16,80	16,23	16,55	17,13
Relativní zrychlení [%]	0,00	38,87	40,96	39,77	42,29
Relativní zrychlení [%] Paralelizovatelná část	0,00	44,57	46,95	45,62	43,20

Tabulka 21: Výsledky testu reálnou simulací – algoritmus dělení mapy na výpočetním systému č.2.

Testovací výpočetní systém č.3 Dual-Core Opteron 2220@ 2.80GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	22,67	13,68	9,46	8,47	7,35
Relativní zrychlení [%]	0,00	39,67	58,28	62,63	67,60
Relativní zrychlení [%] Paralelizovatelná část	0,00	45,30	66,57	71,56	77,20

Tabulka 22: Výsledky testu reálnou simulací – algoritmus dělení agentů na výpočetním systému č.3.

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	22,67	13,77	8,77	8,37	8,19
Relativní zrychlení [%]	0,00	39,26	61,32	63,10	63,90
Relativní zrychlení [%] Paralelizovatelná část	0,00	44,85	70,05	72,06	72,97

Tabulka 23: Výsledky testu reálnou simulací – algoritmus dělení mapy na výpočetním systému č.3.

Testovací výpočetní systém č.4 Xeon X5355@ 2.66GHz

Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	24,30	14,96	10,75	7,70	6,73
Relativní zrychlení [%]	0,00	38,44	55,76	68,32	72,31
Relativní zrychlení [%] Paralelizovatelná část	0,00	44,82	65,02	79,65	84,30

Tabulka 24: Výsledky testu reálnou simulací – algoritmus dělení agentů na výpočetním systému č.4.

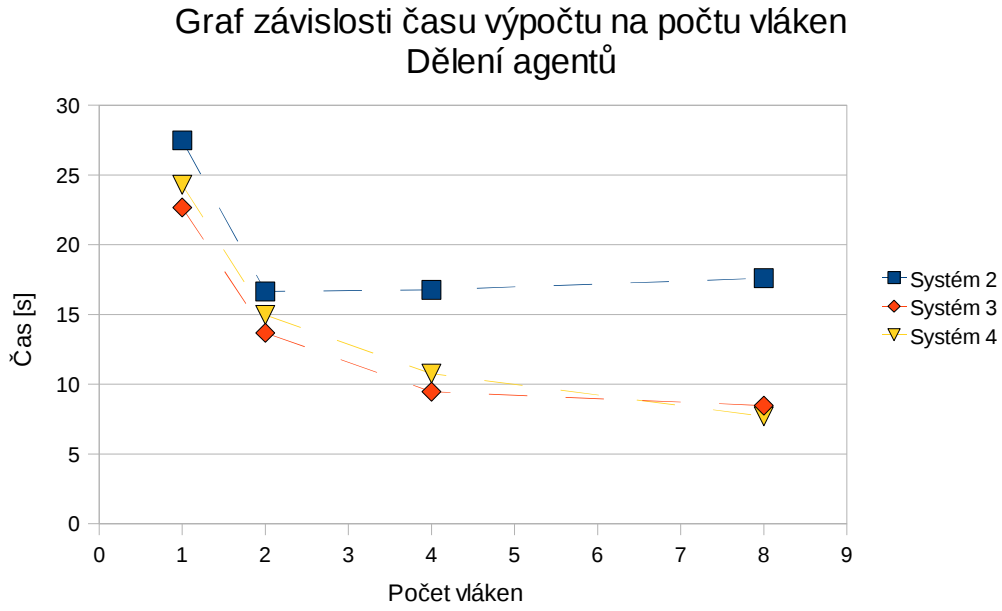
Počet vláken	1 (referenční)	2	4	8	auto
Čas [s]	24,30	15,67	9,32	7,18	6,87
Relativní zrychlení [%]	0,00	35,51	61,65	70,45	71,73
Relativní zrychlení [%] Paralelizovatelná část	0,00	41,41	71,88	82,15	83,63

Tabulka 25: Výsledky testu reálnou simulací – algoritmus dělení mapy na výpočetním systému č.4.

Konfrontace s předpoklady

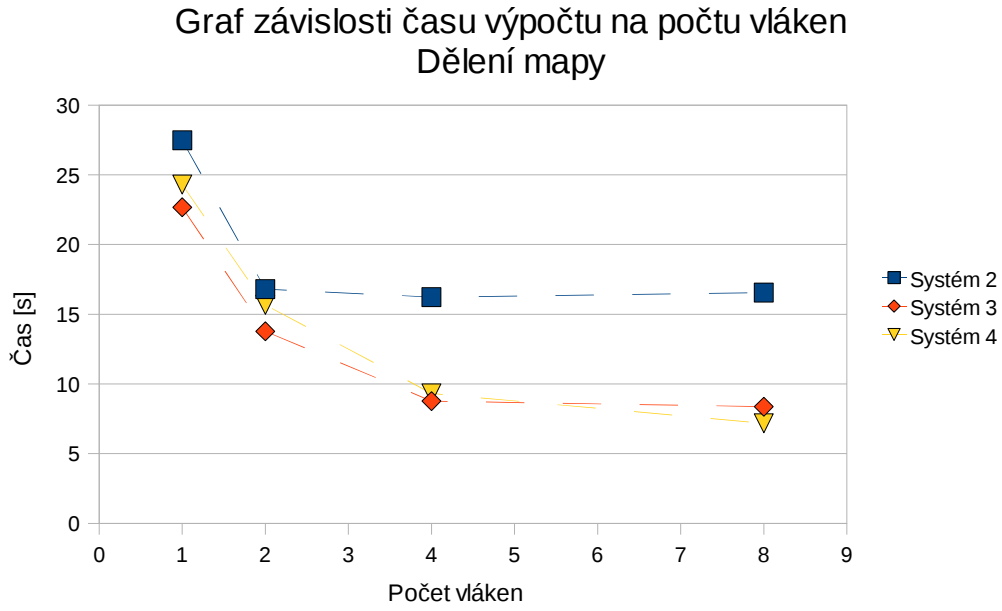
I u tohoto testu se obecně předpokládané teorie potvrdily. Výpočetní systém č.1 dosahuje s rostoucím počtem vláken mírného zpomalení, zatímco ostatní výpočetní systémy zaznamenávají zrychlení odpovídající počtu jejich procesorových jader a počtu vláken výpočtu. Tento předpoklad nebyl porušen ani u jednoho z testovaných algoritmů, jak názorně vizualizují následující grafy.

Konkrétní předpoklad, že výsledky algoritmu dělení agentů budou lepší, než výsledky algoritmu dělení mapy, nelze jednoznačně potvrdit. Zatímco při automatickém dělení do vláken přístup dělení agentů získává mírnou převahu, při pevném počtu vláken dosahuje lepších výsledků přístup dělení mapy. Rozdíly se pohybují v řádu jednotek procent.



Graf 5: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení agentů

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.



Graf 6: Graf závislosti času výpočtu na počtu vláken – algoritmus dělení mapy

Graf neobsahuje údaje výpočetního systému č.1 z důvodu měřítka.

7 Závěr

7.1 Zhodnocení výsledků testů

Testy ve většině případů splnily předpoklady, ale vyskytly se i případy, kdy tomu tak nebylo. Testování jednoznačně potvrdilo, že paralelizace na výpočetním systému s jedním jádrem nepřináší zrychlení, naopak spíše výpočet zpomaluje. Na systémech s více procesorovými jádry představuje paralelizace významný přínos – u systému s osmi jádry při vhodném využití až téměř 75%. Výjimku zde tvoří algoritmus dělení mapy u testu systematického pohybu agentů – test svou povahou u tohoto algoritmu zcela vyřazuje paralelismus.

Pomineme-li tuto anomálii, lze říci, že s rostoucím počtem procesorových jader klesá čas potřebný pro výpočet. Čas však neklesá lineárně. Nejvýraznější zrychlení výpočtu nastává mezi jednovláknovým a dvou-vláknovým zpracováním. Velikost dalšího relativního zrychlení výpočtu se pohybuje velmi zhruba okolo poloviny velikosti předchozího zrychlení. Spojnice trendu času výpočtu (vyznačená v grafech přerušovanou čarou) tak připomíná logaritmickou křivku převrácenou podle osy *x*.

Z testů můžeme vypozorovat také skutečnost, že efektivita přidávání vláken paralelního zpracování má strop na počtu procesorových jader výpočetního systému. Počet výpočetních vláken větší, než je počet procesorových jader systému přináší zrychlení minimální nebo žádné, ale může výpočet i zpomalit. Toto potvrzuje i srovnání s automatickým dělením zátěže mezi vlákna v režii TBB, které dosahuje výsledků srovnatelných s výsledky výpočtu pevně děleného do počtu vláken odpovídajícímu počtu procesorových jader systému.

Dalším aspektem testu je srovnání algoritmů přidělování agentů jednotlivým vláknům. Jedná se o algoritmy dělení agentů, který dělí množinu agentů v simulaci rovnoměrně mezi jednotlivá vlákna, a dělení mapy, který dělí agenty mezi vlákna na základě jejich pozice na mapě, kdy každé vlákno zpracovává určitou oblast mapy. Tady jsou výsledky ne zcela předpokládané. U syntetických testů náhodného a systematického pohybu agentů se výsledky s předpoklady shodují. U testu náhodného pohybu agentů je algoritmus dělení agentů v řádu procent efektivnější, než algoritmus dělení mapy. U testu systematického pohybu agentů je algoritmus dělení mapy prakticky nepoužitelný, vzhledem k tomu, že test v podstatě vyřazuje paralelismus při výpočtu. Tento test poukazuje na větší přizpůsobivost algoritmu dělení agentů.

Poněkud zvláštním případem je ale test reálnou simulací. Tento test má ze všech testů v podstatě nejvyšší váhu, protože se blíží způsobu, jakým bývá simulátor používán v praxi. Zde jsou výsledky obou algoritmů poměrně vyrovnané – v některých případech dosahuje algoritmus dělení mapy lepších výsledků, než algoritmus dělení agentů, a to hlavně u většího počtu vláken. Příčina tohoto jevu nebyla sice jednoznačně potvrzena, ale já osobně se přikláním k teorii, že tato odchylka od předpokladů je způsobena nestejnou složitostí výpočtu skriptů agentů v simulaci. U tohoto testu je agent typu 1 výrazně výpočetně náročnější, než agent typu 0. Po načtení dat testu se v simulaci vytvoří první tisíc agentů typu 0 a další tisíc agentů typu 1. Agenti získávají ID podle pořadí, v jakém byli vkládáni do simulace. Vzhledem k tomu, že algoritmus dělení agentů dělí množinu agentů na části na základě ID, získá první polovina vláken množinu agentů výrazně výpočetně méně náročnou, než druhá polovina vláken. Díky tomu některá vlákna dokončí svou práci rychle a skončí, zatímco jiná běží podstatně déle a je nutné na ně čekat. Tím celková efektivita výpočtu klesá a výpočet se zpomaluje. Oproti tomu algoritmus dělení mapy přiděluje agenty vláknům podle pozice na mapě, která je u tohoto testu ve většině případů náhodná. Díky tomu dochází k poměrně rovnoměrnému i když ne ideálnímu rozložení zátěže mezi vlákna – každé vlákno dostane ke zpracování část agentů typu 0 a část typy 1. Popsaná skutečnost ukazuje, že ani algoritmus dělení agentů v implementované podobně není ideální a může dosahovat horších výsledků. Řešení tohoto problému by bylo proveditelné vyvažováním zátěže na vlákno před každým krokem. Takové řešení je nastíněno v kapitole 7.2 jako jedno z možných budoucích rozšíření.

7.2 Další možnosti rozšíření

Možností, jak rozšířit tuto práci, je hodně. V podstatě závisí na směru, kterým by se měl další vývoj ubírat. Při zachování současného zaměření lze najít několik rozšíření, která zefektivní výpočet a nabídnou další možnosti paralelizace.

Jako první se nabízí paralelizace provádění modifikujících akcí. Zde by bylo nutné zajistit synchronizaci přístupu k mapě a datům jednotlivých agentů. V případě jednotlivých agentů by pravděpodobně dostačovalo použít mutex společný pro operace manipulující s daty agenta. Tím by byl zajištěn výlučný přístup k datům agenta. V případě mapy by už ale mutex zajišťující výlučný přístup nestačil. Takový mutex by totiž způsobil, že veškeré přístupy k datům mapy by byly provedeny výlučně – v konkrétním čase by k mapě mohla přistupovat pouze jedna z nich, což je časově stejně náročné, jako sekvenční provedení akcí. Řešením by bylo použití kontejneru *concurren_hash_map*, který zajišťuje synchronizaci přístupu na podrobnější úrovni. I s těmito prostředky by bylo náročné optimalizovat toto rozšíření natolik, aby představovalo reálný přínos.

Dalším možným rozšířením je paralelizace skriptů agentů a případně skriptů výpočtu statistik a ukončovací podmínky. Přestože LUA obsahuje implementaci vláken, tato implementace je kvaziparalelní – v čase běží vždy jen jedno z vláken. Toto rozšíření by tedy v podstatě vyžadovalo zásah do knihovny interpretu LUA. Jistou možností, jak toto obejít, je přidat do rozhraní agenta a simulátoru funkci, která by jako parametr brala formátovací řetězec, řetězec kódu, který se má provádět a sadu proměnných, se kterými má pracovat a tento kód by provedla paralelně. Toto řešení by ale bylo poměrně neintuitivní a pracné pro uživatele – programátora skriptů agenta a simulátoru.

Posledním rozšířením, které v této práci zmíním, je automatické vyvažování zátěže mezi vlákny. U reálných simulací často nastávají případy, že jsou skripty agentů různě složité (např. Test reálnou simulací – kap. 6.1.3 - rozdíl složitosti skriptu agenta typu 0 a agenta typu 1). V případě, že je pak množina agentů v simulaci rozdělena rovnoměrně na stejně velké části a tyto jsou prováděny ve vláknech, dochází k tomu, že vlákno zpracovávající převážně jednodušší agenty proběhne, v krátkém čase skončí a nečinně čeká, zatímco vlákno zpracovávající převážně složitější agenty provádí výpočet dlouho a zdržuje další běh aplikace. Řešením tohoto problému by byla možnost na základě zátěže jednotlivých vláken měnit množiny agentů zpracovávané jednotlivými vlákny – například změnou rozsahu ID agentů, které má vlákno zpracovávat. Heuristikou může být jednoduše čas provádění jednotlivých vláken – pokud v časech nastane rozdíl větší než definovaný, dojde k přeskupení zpracovávaných množin pro příští krok. Tento postu je použitelný jen pro algoritmus dělení agentů. V případě algoritmu dělení mapy je množina agentů zpracovávaná jedním vláknem dána částí mapy, kterou vlákno zpracovává.

Toto byl tedy nástin možných rozšíření simulátoru. Samozřejmě rozšíření může být víc a mohou se týkat i jiných oblastí, než paralelizace. Pro návrh dalších rozšíření by bylo vhodné využít poznatky z praktického nasazení simulátorů agentních systémů a případně konzultovat další postup s uživateli.

Literatura

[1] *Multiagentní systémy.* [online]. [rev. Květen 21 2008]. [cit. 2009-05-07]. URL:<http://multiagent.tym.cz/nicoolas/Home.php>

[2] Holand John H.: *Hidden order: How adaption builds complexity.* první vydání. Perseus Books, 1996. ISBN 0-201-44230-2

[3] Hepnar Josef: *Simulace anticipačního chování tvorů na bázi umělého života.* 2008. FEL ČVUT v Praze. [online]. [cit. 2009-05-07]. URL:<https://dip.felk.cvut.cz/browse/pdfcache/hepnaj2_2008bach.pdf>

[4] Ippolito Greg: *POSIX thread (pthread) libraries.* [online]. [rev. 2004]. [cit. 2009-05-07]. URL:<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

[5] *Intel Threading Building Blocks 2.1 for Open Source.* [online]. [cit. 2009-05-07]. URL:<http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference%20(Open%20Source).pdf>

[6] *Summary of OpenMP 3.0 C/C++ Syntax* . listopad 2008. [online]. [cit. 2009-05-07]. URL:<http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf>

[7] Leijen Daan, Hal Judd.: *MSDN: Optimize Managed Code For Multi-Core Machines.* [online]. [rev. October 2007]. [cit. 2009-05-07]. URL:<http://msdn.microsoft.com/cs-cz/magazine/cc163340(en-us).aspx>

[8] Haridi Seif a spol.: *Parallel Agent Based Simulation on PC Cluster* . Swedish Institute of Computer Science. [online]. [cit. 2009-05-07]. URL:<http://www.sics.se/~seif/Presentations/simulationIBM.pdf>

[9] Dorigo M., Gambardella L.M.: *Ant colony system: a cooperative learning approach to the travelingsalesman problem.* Evolutionary Computation, ročník 1997, číslo 1: s. 53 – 66. [online]. [cit. 2009-05-07]. URL:<http://www.endaridge.org/publications/conferences/CEC/CEC_2006_Ridge.pdf>

[10] *TinyXml Documentation.* [online]. [rev. August 18 2006]. [cit. 2009-04-19]. URL:<http://www.grinninglizard.com/tinyxmldocs/index.html>

[11] *Lua 5.1 Reference Manual* . [online]. [rev. January 22 2009]. [cit. 2009-05-07]. URL:<http://www.lua.org/manual/5.1/>

[12] Lísal Martin: *Co je to paralelní počítání?.* [online]. [cit. 2009-05-07]. URL:<http://physics.ujep.cz/~mlisal/par_progrm/1_tyden1.pdf>