VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

DYNAMICKÉ VYVAŽOVÁNÍ ZÁTĚŽE V PARALELNÍCH APLIKACÍCH

DYNAMIC LOAD-BALLANCING IN HPC APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. VOJTĚCH DVOŘÁČEK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19646/2016/xdvora0y

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Dvořáček Vojtěch, Bc.**
Obor: Inteligentní systémy
Téma: **Dynamické vyvažování zátěže v paralelních aplikacích**
Dynamic Load-Balancing in Parallel Applications
Kategorie: Modelování a simulace

Pokyny:

1. Seznamte se s architekturou superpočítačů Anselm a Salomon.
2. Osvojte si programovací techniky a potřebné programové prostředky pro implementaci distribuovaných aplikací na superpočítači.
3. Prostudujte současné techniky pro dynamické vyvažování zátěže.
4. Navrhněte ukázkovou aplikaci demonstrující principy dynamického vyvažování zátěže.
5. Implementujte ukázkovou aplikaci s moduly pro identifikaci přetížených míst, redistribuci práce a nutnou synchronizaci.
6. Ověřte chování aplikace na různých scénářích (přetížený uzel, přetížená síť).
7. Zhodnoťte dosažené výsledky a diskutujte přínos pro praxi.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2
L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato práce se zabývá implementací dynamického vyvažování zátěže do paralelního modelu šíření tepla v chladiči procesoru. První část představuje obecně problematiku dynamického vyvažování a současné metody jejího řešení. Zároveň popisuje použitý model a nástroje pro implementaci jako je knihovna MPI pro komunikace nebo HDF5 pro ukládání dat. Dále byl v rámci práce navržen a implementován paralelní simulační model šíření tepla s dynamickou 2D dekompozicí čtvercové výpočetní domény. S touto doménou pracuje geometrický vyvažovací algoritmus, navržený v rámci práce. Implementace dále využívá knihovnu Zoltan pro přenos dat. Simulační model je implementován v C/C++ s využitím MPI komunikací. Na závěr je provedena řada experimentů, které demonstrují dosažený efekt dynamického vyvažování spolu s motivací pro další výzkum v této oblasti.

Abstract

This thesis aims to implement dynamic load balancing mechanism into the parallel simulation model of the heat distribution in a CPU cooler. The first part introduces theoretical foundations for dynamic load balancing, describing current solution approaches. The second part refers to the heat distribution model and related topics such as MPI communications library or HDF library for data storage. Then it proceeds to the implementation of simulation model with dynamic 2D decomposition of square model domain. Custom geometry based dynamic load balancing algorithm was introduced, which works with this decomposition. Important part of the implementation is Zoltan library, used especially for data migration. At the end, a set of experiments was presented, which demonstrates load balancing abilities of designed model together with conclusions and motivation for future research.

Klíčová slova

dynamické vyvažování zátěže, model šíření tepla, MPI, paralelní výpočty, Zoltan

Keywords

dynamic load balancing, heat distribution model, MPI, parallel computation, Zoltan

Citace

DVOŘÁČEK, Vojtěch. *Dynamické vyvažování zátěže v paralelních aplikacích*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jiří Jaroš, Ph.D.

Dynamické vyvažování zátěže v paralelních aplikacích

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Vojtěch Dvořáček
24. května 2017

Poděkování

Rád bych zde poděkoval Ing. Jiřímu Jarošovi, Ph.D. za svědomité vedení této práce, za čas, který mi věnoval, přátelský přístup a mnoho cenných rad.

Zároveň bych rád poděkoval Lukinovi a Raďasovi za jejich pečlivou korekturu textu.

Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy v rámci projektu velkých infrastruktur pro výzkum, experimentální vývoj a inovace - „IT4Innovations národní superpočítačové centrum – LM2015070“.

Obsah

Úvod **3**

1 Dynamické vyvažování zátěže **5**
1.1 Obecný pohled 5
1.2 Statické vyvažování zátěže 6
1.3 Dynamické vyvažování zátěže 6
1.3.1 Centralizovaný a distribuovaný přístup 8
1.3.2 Současné metody 8

2 Model šíření tepla **10**
2.1 MPI - Message Passing Interface 10
2.1.1 Typy MPI komunikací 11
2.1.2 Problémy při komunikaci 11
2.2 Popis modelu 12
2.2.1 Numerická metoda 13
2.2.2 Výměna dat a halo zóny 13
2.2.3 Datový formát HDF5 13
2.3 Implementace statické verze 14
2.3.1 Detailní popis modelu 14
2.3.2 Třída BlockDescriptor 15

3 Příklad nevyváženého stavu **18**
3.1 Hardware pro experimenty 18
3.2 Softwarové vybavení 19
3.2.1 Experimenty 20
3.2.2 Výsledky měření 20
3.2.3 Vizualizace zpomalení 22

4 Návrh řešení **25**
4.1 Vysokoúrovňový popis 25
4.1.1 Načítání vstupních dat 26
4.1.2 Měření 27
4.1.3 Rozhodování 27
4.1.4 Výpočet nové dekompozice 28
4.1.5 Aktualizace dat 28
4.2 Hierarchie tříd 29
4.3 Měření výkonu 30
4.4 Detekce nevyvážení 30

4.5 Zoltan . . . 30
4.5.1 Metody vyvažování . . . 31
4.5.2 Zoltan API . . . 32

5 Implementace **33**
5.1 Dekompozice domény . . . 33
5.2 Komunikace sousedních bloků . . . 34
5.2.1 MPI komunikátor . . . 34
5.2.2 MPI skupiny . . . 35
5.2.3 Princip komunikace . . . 35
5.2.4 Halo zóny . . . 36
5.3 Migrace dat mezi procesy . . . 36
5.3.1 Integrace knihovny Zoltan . . . 36
5.3.2 Rozhraní Zoltanu pro migraci dat . . . 38
5.3.3 Zdrojová a cílová pole . . . 38
5.3.4 Výpočet seznamů pro migraci . . . 39
5.4 Dynamické vyvažování . . . 39
5.4.1 Proporcionální dělení . . . 39
5.4.2 Obnova rovnoměrného rozdělení . . . 41
5.5 Rozhraní pro simulaci . . . 41
5.5.1 Validita modelu . . . 42
5.6 Vývoj a dokumentace . . . 42
5.6.1 Veřejný repozitář . . . 43
5.7 Kompilace a spuštění . . . 43

6 Experimenty **44**
6.1 Migrace dat . . . 44
6.1.1 Metoda . . . 44
6.1.2 Anselm . . . 45
6.1.3 Salomon . . . 46
6.2 Dynamické vyvažování . . . 46
6.2.1 Metoda . . . 46
6.2.2 Ideální případ . . . 47
6.3 Zhodnocení výsledků . . . 50

Závěr **52**

Literatura **53**

A Překlad Zoltanu **55**

Úvod

Oblast vysoce výkonného počítání se v posledních letech rychle rozvíjí. Žebříčky nejvýkonnějších superpočítačů se mění každým rokem a systémy, využívající masivního paralelizmu ve vysoce náročných aplikacích jsou stále dostupnější. Výstavba nových výpočetních center je vždy spojena s obrovskými investicemi. Přestože žebříčky hodnotí tyto systémy především podle instalovaného výpočetního výkonu, mnohem důležitější, ale také hůře měřitelná je schopnost, takový výkon efektivně využít. Nároky na efektivitu kódu jsou proto i jednou z hlavních podmínek, na základě kterých jsou výpočetní prostředky uživatelům poskytovány.

Rostoucí výpočetní výkon umožňuje řešení stále složitějších problémů. Jednou z oblastí, která produkuje jedny z nejnáročnějších aplikací je oblast simulačních modelů, především těch zaměřených na přírodovědné či medicínské aplikace. Tato kategorie zahrnuje mimojiné spojité simulační modely s vysokými nároky na přesnost výpočtů v plovoucí desetinné čárce. Paralelizace takových modelů je často jedinou cestou jak dosáhnout požadovaných výsledků v přijatelném čase, což je klíčová vlastnost pro jejich konkurenceschopnost.

S přechodem do paralelního výpočetního prostředí přichází, přes jeho nesporné výhody, také mnoho komplikací spojených s režií paralelních algoritmů. Kromě samotných výpočtů se do popředí dostává potřeba efektivně využívat také prostředky komunikační. Ty jsou výrazně pomalejší a jejich vliv na rychlost výpočtu paralelních aplikací tak nabývá zásadní důležitosti.

Oproti sekvenčnímu prostředí, kde pracujeme s operační pamětí, vstup/výstupními operacemi nebo komunikacemi po síti odděleně v čase, umožňuje paralelní prostředí souběh těchto událostí. To vytváří prostor pro vznik stavů, kdy je celý výpočet zpomalen nebo zcela dočasně zastaven čekáním na potřebnou synchronizaci s periferiemi, nebo jednotlivými procesy, které jsou z různých důvodů zpomaleny vnějšími vlivy. Takové situace se snaží řešit modely, využívající vyvažování zátěže.

Smyslem dynamického vyvažování zátěže je monitorovat stav systému během výpočtu a detekovat nebo přímo predikovat stavy, které způsobují latence a provést nezbytné akce k minimalizaci jejich dopadu, nebo jejich úplné eliminaci. Zároveň je třeba vždy hledat vhodný kompromis mezi délkou efektivního výpočtu a režií spojenou s dynamickým vyvažováním. Silnou motivací však zůstává schopnost zefektivnit výpočet, což v důsledku vede jak k časovým tak ekonomickým úsporám. Obzvlášť silnou motivací jsou potom medicínské aplikace, kde je v sázce lidské zdraví nebo výzkum s ním spojený.

Cílem této práce je prostudovat současné přístupy a metody dynamického vyvažování zátěže, tyto přístupy analyzovat a na základě poznatků navrhnout a implementovat dynamické vyvažování nad vybraným modelem. Takový systém by měl zahrnovat především nástroje pro detekci přetížených částí simulace a subsystém, který bude na základě naměřených dat provádět samotné vyvažování.

V poslední kapitole je popsána řada experimentů se zaměřením na výkon navrženého modelu a efektivitu implementovaného vyvažování v modelových situacích.

Závěrem práce diskutuje dosažené výsledky a navrhuje další zlepšení. V neposlední řadě práce poskytuje robustní základ pro další výzkum v této oblasti v rámci doktorského studia.

Tato diplomová práce navazuje na semestrální projekt, v rámci něhož byla implementována ukázková statická verze modelu spolu s teoretickými základy z oblasti vyvažování popsaná v prvních třech kapitolách.

Kapitola 1

Dynamické vyvažování zátěže

Cílem této úvodní kapitoly je nahlédnout do problematiky vyvažování zátěže (angl. load balancing), a to především toho dynamického. Oblast vyvažování zátěže je však obecně širší. Nyní se pokusím nadefinovat základní pojmy a porovnat vlastnosti jednotlivých přístupů.

1.1 Obecný pohled

Jak bylo řečeno v úvodu, přechod vysoce náročných aplikací ze sekvenčního do paralelního prostředí přinesl kromě mnoha výhod také mnoho problémů s tímto prostředím spojených.

Abychom mohli paralelizovat nějaký výpočet, musíme být schopni rozdělit jeho výpočetní doménu na několik nezávislých celků – subdomén. Jejich počet by měl odpovídat počtu dostupných procesorů, nebo být případně vyšší, abychom využili potenciál paralelního zpracování. V dalším textu uvažuji případ, kdy je každý paralelní proces mapován na jeden procesor, což je ve většině případů optimální. Slova proces a procesor pro tento případ definujme jako synonyma, pokud není uvedeno jinak. Pro dosažení kvalitního rozdělení, musíme vytvořit vhodný model, který jednoznačně charakterizuje výkon jednotlivých procesorů, případně celých uzlů.

Takový model bude zřejmě sledovat především:

- výkon procesoru
- velikost paměti
- propustnost a odezvu I/O a komunikačního rozhraní
- topologii propojení uzlů/procesorů

S tím koresponduje samotný problém vyvažování, na který lze nahlížet jako na optimalizační problém. Ten můžeme řešit podle několika kritérií, především pak na základě:

- Přidělené práce.
- Času meziprocesové komunikace.
- Času výpočtu.

Jednotlivá kritéria můžeme dále zároveň kombinovat, což může vést k lepším výsledkům. Podle toho kdy a jak výpočet vyvažujeme můžeme problematiku rozdělit na dva základní přístupy - statické a dynamické vyvažování[15].

1.2 Statické vyvažování zátěže

Základní myšlenkou statického vyvažování je, že veškerá rozhodnutí o rozdělení domény provedeme staticky, před zahájením výpočtu. Na základě aktuálního stavu systému, nebo statistik posbíraných během provozu určíme rozdělení domény, které se během výpočtu nemění. Výhodou takového přístupu je nulová režie během výpočtu, kdy není potřeba přesouvat přidělené subdomény mezi procesory. V některých případech [21] je taková režie za běhu nežádoucí a statické vyvažování může vést k uspokojivým výsledkům.

Hlavní nevýhodou tohoto přístupu je samozřejmě fakt, že nijak nereflektuje změny v chování systému během výpočtu. Ty mohou být zásadní i vzhledem k tomu, že aplikací z této oblasti často běží několik dnů, v extrémním případě i týdnů. Proto nebude statický přístup dostačující.

Semi-statické vyvažování zátěže

Tento přístup představuje kombinaci obou výše uvedených přístupů. Jedna analýza je provedena staticky před spuštěním aplikace pro prvotní rozdělení a za běhu se pak vyvažuje dynamicky.

1.3 Dynamické vyvažování zátěže

Základem dynamického vyvažování je monitorování programu během výpočtu a následná reakce na případné nevyvážené stavy.

Celý proces můžeme rozdělit do několika kroků:

1. Měření aktuálního stavu.
2. Detekce nevyváženosti.
3. Rozhodování o přerozdělení.
   (a) Přerozdělení domény a přesun dat.
   (b) Ponechání aktuálního stavu.

V případě, že nastane situace, kdy není rozložení optimální, měl by systém tuto situaci detekovat, vyhodnotit její závažnost a potenciální dopad a pokud jsou splněny modelované předpoklady, tak iniciovat nové mapování domény mezi jednotlivé procesory. Žádný z těchto kroků není triviální a k nalezení vhodného algoritmu je třeba řada znalostí a experimentů. Zároveň si musíme uvědomit, že tyto kroky jsou prováděny iterativně v reálném čase za běhu naší aplikace. Je tedy potřeba dbát na to, aby samotná režie vyvažování nevyžadovala významnou část výpočetního času. Předpokládáme, že časově nejnáročnější je přerozdělení domény. Samotné sbírání dat a rozhodování by mělo být v ideálním případě zanedbatelné.

Obecně můžeme říct, že hledáme kompromis mezi optimálním rozložením výkonu a množstvím potřebných komunikací.

Měření aktuálního stavu

V této fázi se snažíme posbírat co nejvíce směrodatných informací o stavu systému, na kterém běží náš kód. Informace mohou být různorodé, jak hodnoty, které jsme experimentálně naměřili během výpočtu, tak data poskytnutá operačním systémem. S ohledem

na povahu sledované aplikace se pak snažíme vybrat takové hodnoty, které by našemu rozhodovacímu modelu poskytly co největší přesnost. Typicky tedy budeme uvažovat kombinaci několika různých veličin.

Jednoduchým příkladem může být měření doby výpočtu u klíčových operací ve sledované aplikaci, ze kterých vytvoříme například agregovanou hodnotu a tu budeme iterativně posílat procesoru, který provádí rozhodování – tzv. kolektor.

Dalším příkladem takových dat mohou být systémem posbírané statistiky o vytížení periferií, které sledovaná aplikace často využívá.

Detekce nevyváženosti

V této fázi je cíl jasný. Analýzou dostupných dat zjistit, jestli lze aktuální stav považovat za nevyvážený. V tomto bodě je důležité především definovat správné indikátory a mezní hodnoty. Ty zjistíme například řadou experimentů nad definovaným matematickým modelem.

Mezi hlavní příčiny nevyváženosti v paralelních systémech patří[16]:

- Nevhodné rozdělení domény mezi procesory, vzhledem k jejich výkonu.
- Vstupně/výstupní operace.
- Čekací stavy nezbytné pro synchronizaci komunikace mezi procesy.
- Vnější vlivy při běhu v nededikovaném prostředí.

První případ se projeví především při práci v nehomogenních clusterech, kde jednotlivé výpočetní uzly nemají stejnou HW konfiguraci, nebo ji mají, ale frekvence procesoru se mění např. kvůli nedostatečnému chlazení. V případě rozdílné HW konfigurace je problém samozřejmě řešitelný i statickým vyvažováním před samotným spuštěním výpočtu.

Druhý případ se týká komunikačního rozhraní. Pokud dojde k prodlevám v doručení zpráv vlivem pomalejšího výpočtu nebo vytížené komunikační linky, dojde k čekání na dokončení nezbytných komunikací. *Čekací stav* tedy můžeme definovat jako čas, kdy je jeden nebo několik procesorů v nečinném stavu a čekají na dokončení výpočtu, který provádí ostatní procesor/procesory.

Konečně pokud pracujeme v clusteru, ve kterém sdílíme prostředky s ostatními uživateli. Situace se může za běhu měnit velmi dramaticky, pokud jsme nuceni sdílet kapacitu sítě a vstup/výstupních operací. Reálně může být samozřejmě vlivů více.

Rozhodování o přerozdělení

V této fázi dochází k zásadnímu rozhodnutí ovlivňujícímu úspěšnost metody. Je proto vhodné jí věnovat zvláštní pozornost. Vzhledem k tomu, že přemapování domény může být velmi drahé, je správné vyhodnocení nevyváženosti kritické.

Při rozhodování řešíme optimalizační problém, kde na jedné straně stojí ztráty efektivity, způsobené vzniklým nevyvážením a na straně druhé cena přemapování, která zásadně ovlivní celkovou dobu výpočtu. Pokud budeme přemapování provádět velmi často, bude výrazně prodlužovat dobu výpočtu a celý mechanizmus potom ztratí smysl.

Dalším problémem je krátkodobé zpomalení, které může být významné, ale trvat třeba jen jednu iteraci. Pokud bychom jej začali okamžitě vyvažovat, řešíme problém, který už neexistuje. Rozhodování je proto vhodné založit na více měřeních.

Optimální řešení by mělo volit frekvenci přemapování tak, aby se zkrátila celková doba výpočtu oproti stavu, kdy systém ponecháme nevyvážený po celou dobu běhu. Rozhodovací model proto musí být schopen co nejpřesněji odhadovat cenu přemapování a predikovat zpoždění vzniklé aktuálním nevyváženým stavem. Takový odhad bude závislý na dostatku informací o aplikaci, kterou vyvažujeme. To je jedním z důvodů, proč je velmi obtížné vytvořit univerzální vyvažovací model.

Přerozdělení domény a přesun dat

Abychom dokázali přerozdělit výpočetní doménu, musí proběhnout komunikace s přeposláním dat mezi několika nebo dokonce všemi procesory. V takovém případě je kritická cena přenosu. Pokud by režie přemapování trvala déle, než samotná periferní operace, ztrácela by smysl. Zvláště u systémů, kde by se přemapování často opakovalo, by se mohlo nakumulovat značné zpoždění.

Prakticky lze problém rozdělit do dvou kroků:

1. Serializace a přenos doménových dat.
2. Změna metadat jednotlivých subdomén.

V prvním kroku musíme přesunout data, která využívají jednotlivé bloky k výpočtu. Navíc musíme počítat se změnou velikosti a tvaru subdomény. Prvním krokem je tedy vhodná serializace dat, která umožní jejich snadný přenos po síti. Pokud navíc měníme i velikost nebo tvar, je nutné zajistit, aby přidělený proces, obdržel data z několika zdrojů a byl je schopen de-serializovat v jeden celek, použitelný k výpočtu daným algoritmem.

Změna metadat pak musí zajistit obnovení konzistence na jednotlivých procesorech. Systém musí být především schopen, změnit nastavení komunikace mezi subdoménami, vzhledem k tomu, že sousední subdomény budou nyní náležet jiným procesorům. Změnou tvaru subdomény se změní také počet procesorů, což musíme řešit v modelech, kde mezi procesory existují datové závislosti.

1.3.1 Centralizovaný a distribuovaný přístup

Jedním z význačných parametrů systému dynamického vyvažování je to, kdo provádí ono sbírání dat a následné rozhodování. Tady můžeme rozlišovat centralizovaný nebo distribuovaný přístup [15].

U centralizovaného přístupu, provádí sbírání a celý další rozhodovací proces jediný procesor – arbitr – například root v MPI. To je poměrně intuitivní přístup, ale může vzhledem k iterativní povaze problému vést k přetěžování jednoho uzlu jak po stránce komunikace, tak ceny výpočtů. Vzniká tak nevyvážení, které je potřeba kompenzovat.

Distribuovaný přístup se na druhé straně snaží rozložit doménu na menší autonomní celky, kde opět existuje proces ve funkci arbitra, který sbírá data jen ze svého okolí, což vede především v masivně paralelních aplikacích ke snížení komunikačních nároků na daný procesor. Na rozdíl od centralizovanému přístupu nemá arbitr přehled o celém systému, takže vyvažování nemusí dosahovat v některých případech tak dobrých výsledků.

1.3.2 Současné metody

Pro dynamické vyvažování zátěže existuje v současné době řada metod. Ty se dají v zásadě rozdělit do tří kategorií.

1. geometrické
2. grafové
3. hypergrafové

Geometrické metody jsou vhodné především pro modely, které pracují v nějakém souřadném systému. Tyto metody pak dělí prostor domény na menší části, které přidělují jednotlivým procesům. Patří mezi ně například RCB (Recursive Coordinate Bisection)[2], případně velmi podobná RIB (Recursive Inertial Bisection) [20], nebo HSFC (Hilbert Space Filling Curve) [14]. Tyto metody implementuje například knihovna Zoltan [6], která byla použita i v rámci této práce. Souhrnně lze říci, že tyto metody provádí datovou dekompozici pomocí dělení práce ve výpočetní doméně.

Společným rysem zbylých dvou kategorií je řešení problému pomocí grafů. Sledované veličiny ve vyvažovaném sytému jsou nejprve převedeny na grafovou reprezentaci a vyvažování pak představuje problém dělení grafu [13]. Tento problém je NP-úplný, takže se využívají heuristiky, které jsou předmětem výzkumů. Grafem lze modelovat například jednotlivé uzly jako vrcholy a oceněné hrany mohou představovat cenu komunikace. Grafové algoritmy implementují především knihovny jako ParMETIS [17] nebo PT-Scotch [5]. Zobecněním tohoto přístupu jsou hypergrafové metody. Hypergraf je zobecněním obyčejného grafu, který umožňuje jednotlivým hranám spojovat více vrcholů [3]. Díky tomu má větší modelovací schopnosti a může tak vést k optimálnějším řešením [4].

Vzhledem k tomu, že základem pro tuto práci je model šíření tepla (viz. kapitola 2) který pracuje nad čtvercovou doménou, jako vhodná cesta byl zvolen geometrický přístup. Původním záměrem bylo použít metodu RCB, ale v průběhu se ukázalo, že nevyhovuje některým požadavkům. Byla proto vyvinuta vlastní metoda, založená na geometrickém dělení domény.

Kapitola 2

Model šíření tepla

Jako základ pro experimenty s dynamickým vyvažováním v této práci byl vybrán model šíření tepla v chladiči. Jedná se o jeden z jednodušších modelů, který ale dobře demonstruje problémy vznikající v paralelním prostředí a je tak vhodným odrazovým můstkem pro další vývoj.

Předtím, než se budeme věnovat samotnému modelu, je potřeba uvést několik základních faktů o komunikační knihovně MPI, která tvoří základ výpočtů v paralelním prostředí.

2.1 MPI - Message Passing Interface

Většina současných paralelních aplikací využívá pro meziprocesovou komunikaci některou z implementací rozhraní MPI. Jedná se o standard, poprvé vydaný v roce 1994 [7], který specifikuje rozhraní meziprocesové komunikace pomocí zasílání zpráv. Standardní API je definováno pro jazyky C a Fortran. Hojně se využívají i objektové varianty pro C++, o jejichž standardizování se vedou rozsáhlé debaty v MPI komunitě. Vzhledem k návaznosti na tyto nízkoúrovňové jazyky je základem zprávy v MPI úsek paměti – typicky pole – obsahující serializovaná data. Typy jsou buď primitivní (int, float atp.) nebo lze pomocí MPI volání definovat složitější typy nad těmi primitivními včetně struktur. Jednotlivé procesy nemají sdílenou paměť. Data jsou voláním předána knihovně MPI, typicky ukazatelem, a ta řeší samotný přenos a jeho synchronizaci.

Význam uživatelských datových typů se projevuje právě v případech, kdy je potřeba realizovat výpočty nad sdílenou doménou, která je tvořena například n-dimenzionálním polem. Uživatelské typy umožňují předat MPI část sémantiky dat, díky čemuž může knihovna používat velmi efektivní komunikace - především kolektivní.

Jednou z klíčových vlastností MPI pro použití v distribuovaném prostředí, je zajištění kompatibility napříč architekturami. Knihovna řeší problematiku velikosti typů i endianitu. Díky tomu lze bezpečně provádět výpočet v clusteru na několika architekturách zároveň. Nevýhodou standardních implementací je poněkud zastaralé 32-bitové adresování, což komplikuje především kolektivní komunikace s velkými zprávami omezením velikosti bufferu na 2^{32}.

2.1.1 Typy MPI komunikací

MPI rozlišuje několik druhů komunikací, jednak podle počtu účastníků a tak především podle synchronizace.

- počet účastníků - point-to-point nebo kolektivní
- synchronizace přenosu - blokující, neblokující, synchronní

Point-to-point a kolektivní komunikace

Zatímco komunikace typu *point-to-point* jsou tvořeny jednoduchou výměnou zpráv mezi dvěma procesy, *kolektivní* komunikace jsou uzpůsobeny pro hromadnou výměnu dat. Kolektivních komunikací se mohou účastnit jak všichni účastníci systému, tak jejich různé podmnožiny. Ty lze rozlišit buď zavedením různých značek (tagů) v hlavičce zprávy, nebo efektivněji pomocí komunikátorů. Komunikátory rozlišují podmnožiny procesů na úrovni knihovny a není tedy potřeba explicitní programová obsluha všech volání.

Nejdůležitějšími kolektivními komunikace jsou `broadcast`, `scatter`, `gather` a jejich varianty. Zatímco broadcast rozešle stejná data všem v rámci komunikátoru, `scatter` umožňuje rozptýlit velký datový blok po částech mezi procesory a `gather` jej naopak zpátky spojí.

MPI navíc umožňuje provádět i základní aritmetické operace typické pro paralelní prostředí, jako jsou reduce nebo scan.

Další silnou stránkou kolektivních komunikací je možnost efektivní distribuce i složitějších datových struktur mezi procesy díky definování vlastních datových typů. Ty umožňují pomocí nastavení offsetů efektivně namapovat výpočetní doménu mezi jednotlivé procesy.

Blokující, neblokující, synchronní

Zatímco počet účastníků typy komunikací jednoznačně rozděluje, obě tyto kategorie komunikací mohou být realizovány více způsoby. První dvě kategorie fungují tak, jak bychom očekávali. V případě blokujících je výpočet pozastaven, dokud není komunikace ukončena[1]. U neblokujících variant se volání ihned vrátí, ale potvrzení ukončení komunikace musíme vyřešit dalším voláním. Synchronní volání je blokující, ale navíc zajišťuje, že adresát zprávu přijal (dokončil volání recv).

2.1.2 Problémy při komunikaci

Při implementaci MPI komunikací se vzhledem k použitému typu synchronizace můžeme setkat s několika problémy. Jedním z nejtypičtějších je *deadlock*. Pokud použijeme implicitní volání `MPI_Send`, je ponecháno na konfiguraci systému, zda se použije neblokující (bufferovaná) varianta nebo synchronní alternativa. Hranice je vzhledem k efektivitě přenosu dána velikostí zprávy. U menších zpráv v řádech stovek kB se obvykle používá bufferovaná verze, zatímco pro větší zprávy se jako buffer přímo použije předaná paměť a provede se synchronní varianta. To může vést k těžko odhalitelnému deadlocku. Typickým příkladem takové situace je komunikace procesů v kruhové topologii, kde každý posílá zprávu oběma sousedům. V těchto případech je potřeba explicitně použít neblokující variantu.

Mezi další omezení patří již zmíněná velikost přenášených zpráv, která je limitována 32-bitovým adresováním.

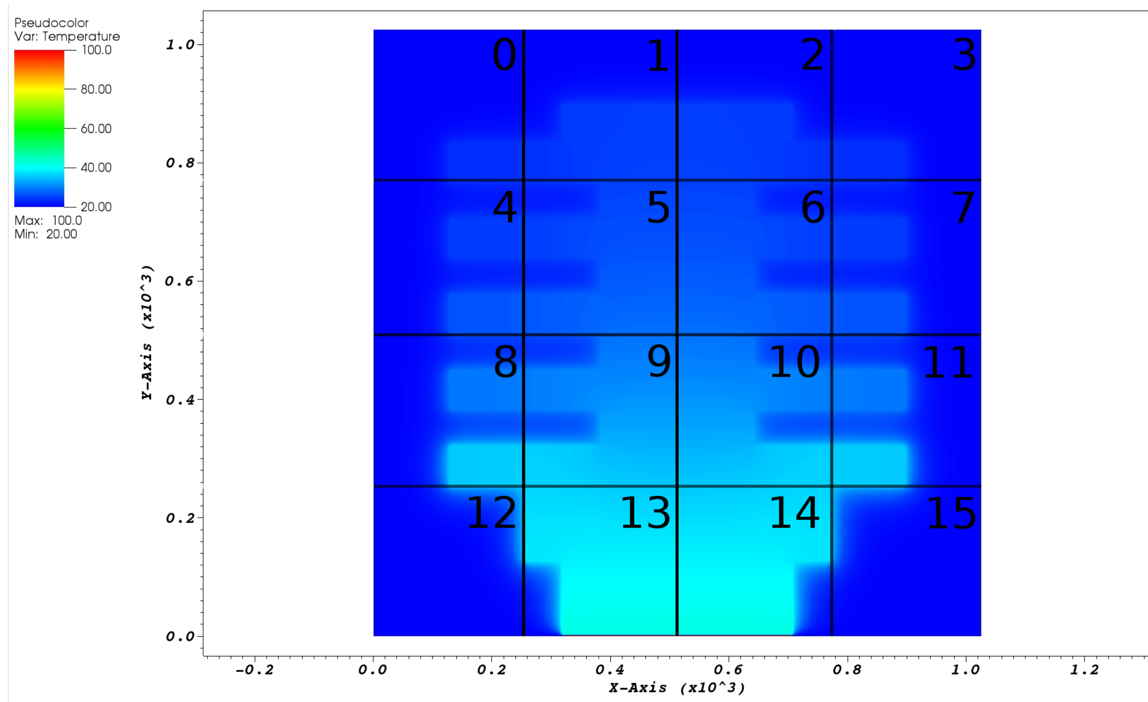
[1] Ukončení komunikace je stav, kdy je opět bezpečné pracovat s pamětí obsahující zprávu

2.2 Popis modelu

Model šíření tepla je simulační model, který počítá odvod tepla od zdroje (procesoru) - ve vertikálním 2D řezu chladičem. Výpočetní doména je tvořena dvourozměrnou maticí, jejíž každý prvek, představuje určitý druh materiálu mající nějakou teplotu. Chladič je klasické věžovité konstrukce, kde základnu tvoří styčná měděná plocha přiléhající k procesoru a teplo je odváděno mezi hliníkové žebrování, které je chlazeno proudícím vzduchem. Na začátku jsou dány počáteční teploty zdroje tepla a chladícího média – vzduchu, který proudí okolo chladiče a mezi jeho žebrováním, čímž jej ochlazuje.

Řez chladičem je diskretizován do pravidelné mřížky teplotních bodů. O každém bodě dále víme, jaký materiál reprezentuje - tedy vzduch, hliník nebo měď. Pro body, které reprezentují vzduch máme definován koeficient perfůze, jenž určuje rychlost proudění vzduchu, a tedy ochlazování okolního materiálu.

V rámci práce pracujeme s 2D dekompozicí výpočetní domény. Následující obrázek ilustruje dekompozici pro 16 procesorů. Obrázek 2.1 je příkladem výstupu modelu [2].



Obrázek 2.1: Příklad dekompozice domény

Čísla představují přidělené MPI ranky. Číslují se vždy od levého horního rohu. Barva odpovídá teplotě v bodě, podle legendy vlevo. V případě, kdy rozdělujeme doménu mezi lichý počet procesorů, se spojí vždy 2 ranky pod sebou v jeden celek, čímž vzniknou obdélníkové bloky. Pokud by byla velikost domény 2048 bodů, měl by tedy každý rank přidělen blok o velikosti 512*512 bodů.

[2] Ve skutečnosti je obrázek převrácen podle osy X, takže rank 0 je rank 12. Zavedená konvence ale pracuje s číslováním od levého horního rohu.

2.2.1 Numerická metoda

Simulační model využívá k výpočtu metodu konečných diferencí v čase - FTDT (Finite Difference Time Domain). Metoda pracuje v diskrétních časových krocích, kde v každém kroku t prochází matici bodů a aproximuje z dostupných údajů teplotu v čase $t+1$. Metoda využívá aritmetický průměr z 4+4 okolí bodu k určení průměrné teploty a následně simuluje odvod tepla proudícím vzduchem.

Pro bod i, j definujeme následující vztah pro odvození teploty:

$$T_{t+1}[i][j] = \frac{T_t[i-1][i] + T_t[i-2][j] + T_t[i+1][j] + T_t[i][j-1] + T_t[i][j-2] + T_t[i][j+1] + T_t[i][j+2] + T_t[i][j]}{9.0} \quad (2.1)$$

Počítáme tedy aritmetický průměr z celkem devíti polí. Čtyři v horizontálním okolí, čtyři ve vertikálním okolí a samotný bod i, j. Tím dostáváme odhad teploty v daném bodě i, j v čase $t+1$ odvozený z času t. Abychom byli schopni simulovat ochlazení, tak musíme získanou teplotu navíc upravit následujícím vztahem:

$$T_{t+1} = \alpha * T_0 + (1-\alpha) * T_{t+1}([i][j]) \quad (2.2)$$

kde $\alpha * T_0$ představuje chlazení médiem. α je relativní rychlost proudění a T_0 je teplota proudícího vzduchu. Druhá část pak představuje zbytek neodvedeného tepla v daném bodě. K výpočtu následující teploty tedy navíc potřebujeme znát typ materiálu a jeho tepelnou vodivost.

2.2.2 Výměna dat a halo zóny

Důležitým prvkem, který musíme posoudit u konkrétních aplikací, je potřeba výměny dat mezi procesory během výpočtu. Z výše popsané metody vyplývá, že se musíme vypořádat s výpočtem okrajových bodů, kde nám okolní body chybí. Subdoménu pak musíme o tyto informace rozšířit. Oblasti, kde dochází k výměně dat s okolím se nazývají *halo* zóny.

Z toho plyne další kritérium pro rozdělení domény a tím je minimalizace halo zón, která vede k menšímu počtu komunikací, a tudíž zrychlení výpočtu. Obzvlášť velký vliv mohou mít halo zóny v případech více dimenzionálních domén spolu s velkými objemy dat. Pokud budou subdomény nepravidelné, může být i adresování komunikací komplikované.

2.2.3 Datový formát HDF5

Model pro serializaci dat používá datový formát HDF5 [10], implementovaný stejnojmennou knihovnou. Tento formát je velmi populární ve vědeckých aplikacích. Jedná se o binární soubory umožňující hierarchicky ukládat data.

Data jsou organizována do skupin a datasetů. Skupina je ekvivalentem souborového adresáře a obsahuje buď datasety nebo další skupiny. Dataset zapouzdřuje data a jejich metadata. Ta specifikují datový typ, sémantické informace (rozměry, knihovní informace) a další uživatelské atributy. Pro práci s vícedimenzionálními daty slouží koncept zvaný *Dataspace*. Ten obsahuje prostorové informace o zpracovávaných datech a umožňuje vytvářet jejich mapování do souboru. *Dataspace* se skládá ze dvou komponent - *memspace* a *filespace*, mezi kterými mapování provádí.

2.3 Implementace statické verze

Výše popsaný model poslouží jako základ pro studium dynamického vyvažování zátěže. Cílem této první fáze bylo implementovat paralelní verzi modelu a demonstrovat dopad vzniku nevyváženého stavu na výpočet.

2.3.1 Detailní popis modelu

Protože je model chladiče používán jako jeden z projektů v rámci předmětu ARC na Fakultě informačních technologií VUT v Brně, je řada komponent již implementována a je také implementován kompletní sekvenční algoritmus pro simulaci. Ten zároveň slouží pro základní validaci paralelní verze. Cílem projektu je pak implementovat paralelní verzi a otestovat její efektivitu.

S projektem je zároveň dodán generátor vstupních dat, který vytváří HDF5 dataset, - konkrétně matici počátečních teplot, matici specifikující materiál a matici tepelné vodivosti. Uživatel může navíc specifikovat velikost generované domény a počáteční teplotu chladícího vzduchu a zdroje tepla. Všechna data jsou uložena do jednoho souboru.

Na straně samotného simulačního modelu jsou implementovány moduly pro načítaní těchto dat, se kterými dále pracuje zmíněný sekvenční algoritmus.

Nejprve stručně popíšu již hotové moduly. Z hlediska návrhu projekt využívá částečně objektový přístup. Vstupní data jsou zapouzdřena ve třídě `TMaterialProperties`. Jak název napovídá, tato třída obsahuje všechna vstupní data spojená s modelem. Celkem se jedná o tři pole - `initTemp`, `domainParams` a `domainMap`. První dvě zmíněná jsou typu `float`, a popisují počáteční teploty jednotlivých bodů a tepelnou vodivost. Třetí je typu `int` a popisuje typ materiálu v daném bodě – hliník, měď, vzduch. Jednotlivé body si navzájem odpovídají – tedy všechna tři pole popisují jednu doménu. Posledním klíčovým atributem je velikost samotné domény.

Další modul představuje struktura `TParameters`, která má ovšem v C++ velmi podobné chování jako třída. Má vlastní atributy i funkce. Tato struktura obsahuje metadata o simulaci – tedy převážně zpracované parametry příkazové řádky a několik metod, fungujících jako *getter*.

Klíčové části modelu se nachází v souboru `proj02.cpp`. Zde najdeme funkce pro sekvenční simulaci, především řídící algoritmus, výpočet teploty podle zadaných vzorců a výstup do souboru. Zápis do souboru typu HDF5 je implementován v sekvenční i paralelní verzi.

Cílem je pak implementovat paralelní verzi simulace s použitím poskytnutých funkcí. Abychom toho dosáhli, je potřeba vyřešit několik dílčích problémů:

1. Výpočet topologie pro dekompozici domény mezi procesory.

2. Distribuce inicializačních dat mezi procesory.

3. Výměna dat v halo zónách během výpočtu.

4. Uložení výsledků.

Implementovaná funkce main zajišťuje načtení modelu ze vstupu a zpracování parametrů. Následně se volá funkce `ParallelHeatDistribution`, která má implementovat kompletní paralelní verzi.

Pokud spustíme model v paralelním prostředí, tedy pomocí MPI, obdrží všechny procesory pouze metadata a samotný model načte pouze root proces.

V první fázi je tedy nutné vypočítat topologii a přidělit práci jednotlivým procesorům. Následně pak všem rozeslat data popisující jejich subdoménu.

Vzhledem k tomu, že je cílem projektu vytvořit platformu pro experimenty s dynamickým vyvažováním, byla snaha co nejvíce oddělit metadata a režii od samotného výpočtu. K těmto účelům byla vytvořena třída `BlockDescriptor`.

2.3.2 Třída BlockDescriptor

Tato třída má za cíl řešit veškeré výpočty spojené s režií modelu a oddělit je tak od hlavního výpočtu. Zároveň byl zvolen co nejdynamičtější přístup, aby mohla být snadno rozšiřitelná o prvky spojené s dynamickým vyvažováním.

Instanci této třídy vytváří každý proces a předává jí pouze velikost nerozdělené domény, svůj MPI rank a počet běžících procesorů. BlockDescriptor na základě těchto informací vypočítá velikost subdomény a pozice subdomén pro jednotlivé procesory. Všechny procesory tedy mají stejná metadata o modelu bez nutnosti zasílání zpráv.

Dekompozice domény

Vstupní doména je pro jednoduchost omezena na čtvercový tvar a počet procesorů je limitován na sudé nebo liché mocniny dvou. Z toho vyplývá, že při rovnoměrném rozdělení mohou subdomény nabývat pouze dvou tvarů – čtverec a obdélník. Je-li počet procesorů roven 2^n, kde n je sudé, odvodíme velikost subdomény podle vztahu 2.3

$$B = D \div \sqrt{W} \tag{2.3}$$

kde B je velikost subdomény, D je velikost domény (jedna strana čtverce) a W je počet procesorů.

V případě, že je počet procesorů lichá mocnina dvou, budou mít subdomény obdélníkový tvar a jejich velikost lze odvodit podle rovnice 2.4 a 2.5 .

$$Bx = D \div \sqrt{W * 2} \tag{2.4}$$

$$By = Bx * 2 \tag{2.5}$$

kde Bx je velikost subdomény ve směry osy x a By ve směru osy y.

Dalším krokem v určení kompletní topologie je výpočet sousedních ranků. Číslování procesorů probíhá od levého horního rohu po řádcích až k pravému dolnímu rohu. K určení sousedů je tedy především potřeba určit pozici ranku v doméně. To lze podle vztahů 2.6 a 2.7

$$Px = rank \bmod Sl \tag{2.6}$$

$$Py = (rank - Px) \div Sl \tag{2.7}$$

Px, Py jsou souřadnice bloku a Sl je počet sloupců. Ten je roven $D \div Bx$. Z těchto informací snadno určíme sousední ranky. Z pozice zjistíme počet sousedů / okrajů a sousední ranky dopočítáme.

Distribuce dat

Dalším důležitým krokem před zahájením samotné simulace je distribuce iniciálních dat modelu mezi jednotlivé procesory. Tu lze provést několika přístupy. Komunikačně nejnáročnější z nich je rozeslání všech dat pomocí broadcast. Tento přístup však zasílá redundantní

data a pro větší úlohy je nepoužitelný. Další možností je série point-to-point komunikací pomocí send, kdy jednotlivé části rozešleme po jedné. To je datově efektivnější než broadcast, ale stále poměrně pomalé. MPI nabízí pro tyto účely kolektivní komunikace typu scatter.

Ten umožňuje rozeslat jedním voláním určité části celku, které můžou mít stejnou nebo i různou velikost a mohou být také na různých místech v bufferu. Problém ovšem nastává ve chvíli, kdy chceme distribuovat části dat, které nejsou uloženy spojitě v paměti. Pro tyto účely musíme sáhnout po vlastních datových typech definovatelných v MPI.

Jedním z nich je typ *subarray*, který umožňuje vybrat podmnožinu v obecně n-dimenzionálním poli. Voláním `MPI_Type_create_subarray`, specifikujeme jak velkou oblast z daného pole chceme vybrat včetně offsetu od začátku bufferu. Můžeme tak definovat typ, který přesně popisuje oblast náležící jedné subdoméně. Voláním `MPI_Iscatterv` pak předáme tento typ, a navíc odpovídající posunutí jednotlivých položek v paměti. Podobně na straně příjemce přijímáme tento datový typ, což námi definovaný úsek domény zapíše do připravené paměti. Distribuci tak lze vyřešit jedním voláním.

Protože navíc jednotlivé sub-domény potřebují halo zóny pro výměnu dat se sousedními sub-doménami, specifikujeme obdobný typ pro pole, které má navíc na každé straně 2 body široké okraje. MPI pak pomocí `Iscatterv` zapíše data doprostřed s respektováním okrajů.

Výměna dat v halo zónách

V tuto chvíli již máme data rozdistribuována mezi procesory a můžeme zahájit simulaci. Během simulace je potřeba vyměňovat halo zóny mezi procesory tak, aby odpovídaly aktuální iteraci. Jak již bylo řečeno, samotný výpočet probíhá tak, že prochází matici hodnot a pro každý bod počítá odhad nové teploty s pomocí 4+4 okolí. V krajních bodech tedy musí každá subdoména před výpočtem znát hodnoty okolních bodů z předchozí iterace.

Na začátku každého cyklu, provedeme řadu komunikací, které výměnu zajistí. Opět můžeme využít vlastních definovaných typů. Tentokrát nadefinujeme celkem 8 typů, pro okraje na každé straně. Čtyři z nich značí okraje, které se mají odeslat a další čtyři značí halo zóny, do kterých přijímáme sousední data.

Perzistentní komunikace

Protože výměna halo zón probíhá v každé iteraci se stejnými parametry, byly zde použity perzistentní komunikace. Jedná se o typ komunikací, kde voláním `MPI_*_init` vytvoříme komunikační požadavek (`MPI_Request`) a poté pomocí `MPI_Start` spustíme komunikaci. V MPI se tak nevytváří interní datové struktury stále dokola.

Třída Neighbor

Pro popis komunikace se sousedem byla vytvořena třída `Neighbor`. Ta zapouzdřuje veškeré informace, které jsou pro komunikaci potřeba. BlockDescriptor vytvoří tyto objekty pro všechny sousedy a vrátí je v C++ kontejneru `std::vector`. Ten pak stačí projít, použít data pro vytvoření perzistentní komunikace a uložit požadavky (typu `MPI_Request`). Toto řešení je vhodné především tam, kde neznáme počet sousedů. Takto s ním lze dynamicky pracovat.

Pokud v budoucnu při dynamickém vyvažování dojde k dělení halo zón mezi více sousedů, můžou být například specifikovány další MPI typy, nebo může být třída Neighbor rozšířena o další potřebná metadata.

V současné chvíli se každý objekt typu Neighbor váže na jeden rank, se kterým se má komunikovat. Kromě ranku obsahuje i tagy. Konvence je taková, že každý procesor při odesílání označí zprávu podle směru, kam ji posílá. Při přijímání z daného směru se pak volí tag opačný. Každý procesor volá jeden send a recv pro každý sousední blok. Tím postupně dojde k výměně všech halo zón.

Zápis průběhu výpočtu

Ten realizuje paralelní verze zápisu HDF5. Připravené funkci se předají data a offset daného bloku a knihovna zajistí paralelní zápis. Ten probíhá pouze v zadaných intervalech (podle výrazu $iteracemodinterval == 0$).

Kapitola 3

Příklad nevyváženého stavu

Cílem této kapitoly je ukázat vliv nevyvážených stavů na celkový běh aplikace, a to především na zpoždění výpočtu. Experimenty vychází z předpokladu, že zpomalením jednoho z procesorů, a tedy vznikem nevyváženého stavu dojde oproti ideálnímu stavu ke zpomalení celého výpočtu. Zpomalení by mělo zhruba odpovídat času stráveného na provádění externí operace. Předpoklad vychází z dříve popsaného faktu, že každý blok iterativně vyměňuje data v halo zónách se svými sousedy. Dojde-li tedy ke zpoždění během této výměny, nemohou ani sousední bloky dokončit výpočet včas. To vede k prodloužení každé opožděné iterace, a tedy zpomalení výpočtu.

3.1 Hardware pro experimenty

Experimenty byly prováděny na superpočítačích v rámci centra IT4Innovations. Přestože detailní popis moderních superpočítačových center je mimo rozsah této práce, je nutné na začátek vymezit alespoň základní pojmy. Základem současných superpočítačových center je síť propojených výpočetních uzlů, zvaná *cluster*. Každý uzel obsahuje množství procesorů s přidruženou operační pamětí. Uzel tvoří základní jednotku pro organizaci výpočetních zdrojů. Běží na něm operační systém, je vybaven síťovým rozhraním a propojen pomocí sběrnic v určité topologii s ostatními uzly. Síťová topologie je navržena s důrazem na vysokou spolehlivost, propustnost, nízkou latenci a odolnost vůči poruchám. Vzhledem k těmto potřebám se oproti běžným TCP/IP uplatňují odlišné strategie v řízení komunikace. Například pro zajištění vysoké spolehlivosti se vyslaná data nezahazují, ale při zahlcení nedojde k jejich odeslání. Podstatné je, že topologie a hardwarové vybavení takové sítě, určuje limity pro komunikaci mezi procesy, jakmile používáme více než jeden uzel.

Tato práce využívá výpočetní prostředky národního superpočítačového centra IT4Innovations provozovaného VŠB v Ostravě. To poskytuje dva výpočetní clustery - starší Anselm a novější a výkonnější Salomon. Oba clustery poskytují `login` uzly pro přihlášení a základní operace. Odtud se alokují potřebné výpočetní zdroje a spouštějí úlohy. Správu úloh a front řídí plánovač `PBS`.

Anselm

Anselm sestává z 209 výpočetních uzlů, kde každý obsahuje 16 jader Intel Xeon E5-2670 série Sandy Bridge a 64 GB operační paměti. Uzly jsou propojeny sběrnicí Infiniband s propustností 40Gb/s. Pro ukládání dat jsou k dispozici 2 svazky HOME a SCRATCH s kapacitou 300 a 130TB.

Salomon

Novější ze dvojice clusterů disponuje 1008 uzly, každý osazený 24 procesory Intel Xeon E5-2695 série Haswell-EP o taktu 2.5GHz a 128GB operační paměti. Komunikaci mezi uzly zajišťuje opět sběrnice Infiniband v konfiguraci 7-dimenzionální hyperkrychle s propustností 56Gb/s. Salomon je dále vybaven 864 akcelerátory Intel Xeon Phi 7120P, každý s 16GB vlastní RAM. Pro ukládání dat je osazen opět dvěma svazky HOME a SCRATCH o kapacitě 500 a 1500TB.

Obrázek 3.1: Superpočítač Salomon, IT4I[1]

3.2 Softwarové vybavení

Na obou clusterech je software organizován do modulů – balíčků poskytujících předinstalovaný software včetně konfigurace proměnných prostředí. Z hlediska experimentů je důležitá správná volba z alternativních balíčků kvůli binární kompatibilitě. Dostupné jsou mj. kompilátory **GNU** a **Intel**, knihovny pro komunikaci **OpenMPI** a **Intel MPI**. Protože oba clustery jsou postaveny na technologiích Intel, je vzhledem k optimalizačním schopnostem preferován Intel překladač a IntelMPI.

Nezbytnou součástí SW vybavení pro tuto práci je také debugger a profiler paralelních aplikací. k těmto účelům poskytuje IT4Innovations hned několik nástrojů.

Jako debugger slouží nástroj **Allinea-DDT**[1]. Jedná se o debugger s grafickým uživatelským rozhraním, který kromě standardních funkcí jako je breakpoint nebo watchpoint

[1] http://www.it4i.cz/portfolio/2/

umožňuje přepínat mezi jednotlivými procesy a dále s nimi pracovat. Zároveň umožňuje efektivně procházet hodnoty proměnných a kontrolovat stav zásobníku.

Pro profilování aplikací slouží další sada nástrojů. Profiler **Score-P** [19] generuje samotná profilovací a trasovací data, která mohou být dále analyzována. Projekt je třeba přeložit pomocí wrapperu překladače a nastavit proměnné prostředí podle potřeby. Spuštěním se pak vygenerují požadovaná data.

Pro základní statistické profilování slouží nástroj **Cube**. Ten vizualizuje nasbírané statistiky podle zadaných metrik - například času stráveného ve funkcích nebo přenesených dat.

Pro pokročilejší výkonnostní analýzu pak slouží nástroj **Vampir**, který zobrazuje data pomocí grafů a umožňuje přesně analyzovat chování jednotlivých procesů.

3.2.1 Experimenty

Pro názornost byly experimenty prováděny na 16 a 64 jádrech, při 100 iteracích výpočtu. Vstupní data tvořila matice modelu o velikosti 2048*2048 bodů. Při dekompozici na 16 jader tedy vznikly bloky o velikosti 512*512 bodů. v případě 64 vznikly analogicky bloky o velikost 256*256 bodů. V případě 16-ti jader nastává specifická situace kdy je výpočet prováděn v rámci jednoho uzlu a je tak minimalizován dopad externích vlivů. Výpočetní uzel je navíc alokován jako dedikovaný, takže nedochází k přepínání kontextu pro další náročné výpočty. Největší vliv mohou mít I/O operace. Byly proto minimalizovány pouze na načtení vstupních dat a dva zápisy průběžného výsledku do souboru – v první a poslední iteraci. Pro zpřesnění výsledku byly průměrovány výstupy ze 3 běhů.

V případě 64 jader, je výpočet prováděn na třech uzlech, kdy v závislosti na architektuře a umístění uzlů může dojít ke zpoždění komunikace. Poměrně náročnou I/O operaci představuje také zápis profilovacích dat (řádově jednotky GB), ale tady lze logicky předpokládat, že přesnost v tomto ohledu řeší profiler.

Tyto podmínky by měly být pro demonstraci problému dostatečné. Při větších vstupních datech jsou kladeny vyšší nároky na komunikace díky adekvátnímu zvětšení halo zón. V případě více jader velikost halo zón klesá, ale náročnější je pak synchronizace komunikací.

Statistiky byly získány pomocí profilovacích dat, nástrojem Cube. Abychom omezili vliv režie, byla počítána pouze doba provádění funkce `RootBehavior` a `OthersBehavior`. Tyto funkce představují hlavní výpočetní smyčku včetně komunikací, distribuce dat a ukládání výsledků.

V obou případech byl zpomalen procesor s MPI rankem 6. V případě 16-ti jader má tento proces 4 sousedy, v případě 64 pouze 3. Tento fakt by neměl mít na výsledek zásadní vliv.

Vytížení bylo simulování voláním sleep, konkrétně `sleep_for` ze standardní knihovny `thread` v C++11. Zpomalení bylo voláno ve 49-ti iteracích, konkrétně v otevřeném intervalu (25, 75) ze 100 iterací a to na 50 a 250ms, což je přibližně dvojnásobek, respektive desetinásobek času jedné iterace. Celkový čas zpomalení tedy byl 2.45 a 12.25 sekund.

3.2.2 Výsledky měření

Nejprve analyzujeme ideální stav - tedy výpočet bez vloženého zpomalení. Provedeme tedy celkem 3 měření výpočtu na 16-ti a 64 jádrech při výše popsané konfiguraci.
Pro obě hodnoty zpomalení byl naměřen předpokládaný čas, tedy 2.45s a 12.25s. První sloupec představuje celkový čas výpočtu pro rank 0, který zároveň provádí načtení a distribuci dat. Druhý sloupec pak představuje součet ostatních ranků.

Měření na 16-ti jádrech

Pro běhy na 16-ti jádrech byly pro ideální případ naměřeny tyto hodnoty [s]:

rank 0	ostatní	celkem	na 1 CPU
35,49	534,15	569,64	35,60
35,08	526,44	561,52	35,09
35,07	526,28	561,35	35,08
Průměr			35,26

Sečtením výpočetního času ranku 0 a ostatních získáme celkovou dobu běhu výpočtu a po vydělení průměrnou hodnotu na jeden proces. Doba vztažená na jeden proces je vhodným ukazatelem celkové doby běhu programu bez režie.
Po přidání zpomalení 50ms byly získány tyto výsledky:

rank 0	ostatní	celkem	na 1 CPU
37,61	564,44	602,05	37,62
37,60	564,23	601,83	37,61
37,63	564,63	602,26	37,64
Průměr			37,62

Pokud budeme vycházet z průměru, vyjde nám zpomalení

$$37,62 - 35,26 = 2,36s$$

Hodnota se tedy blíží 2.45s. To, že ji nepřesahuje, jak bychom očekávali, může být způsobeno např. rozdílnou délkou režie v MPI. Tendence je však jasně patrná i při relativně krátkém zpoždění.
Při zpomalení na 250ms byly pak naměřeny tyto hodnoty:

rank 0	ostatní	celkem	na 1 CPU
47,35	710,42	757,77	47,36
47,35	710,48	757,83	47,36
47,35	710,41	757,76	47,36
Průměr			47,36

Pokud opět porovnáme hodnotu s ideálním případem, dostaneme:

$$47.36 - 35.26 = 12.10s$$

Hodnota opět přibližně odpovídá očekávaným 12.25s.

Měření na 64 jádrech

Na 64 jádrech byly v ideálním případě naměřeny tyto hodnoty:

rank 0	ostatní	celkem	na 1 CPU
26,65	1675,09	1701,74	26,58
26,58	1672,26	1698,84	26,58
426,65	1676,74	1703,39	26,61
Průměr			26,58

Z výše uvedeného je zřejmý dopad většího počtu procesů. Souhrnný čas strávený ostatními ranky je podstatně vyšší, než v předchozích případech. Naproti tomu čas pro rank 0 je nižší, díky menším blokům. V průměru vztaženém na jedno jádro, je pak výpočet kratší o $35.26 - 26.58 = 8.67s$.

Obdobně jako v prvním případě nyní zavedeme zpomalení 50ms po dobu 49 iterací. Dostáváme následující hodnoty [s]:

rank 0	ostatní	celkem	na 1 CPU
29,41	1839,57	1868,98	29,20
28,26	1774,10	1802,36	28,16
29,13	1831,04	1860,17	29,06
Průměr			28,80

a z nich vypočtenou hodnotu zpomalení:

$$28.80 - 26.58 = 2.22s$$

která opět přibližně odpovídá předpokladu. Konečně pro 250ms pak dostáváme tyto hodnoty [s]:

rank 0	ostatní	celkem	na 1 CPU
38,48	2419,30	2457,78	38,40
37,93	2385,34	2423,27	37,86
37,92	2385,63	2423,55	37,86
Průměr			38,04

Zpomalení pak vychází na:

$$38,04 - 26,58 = 11,46s$$

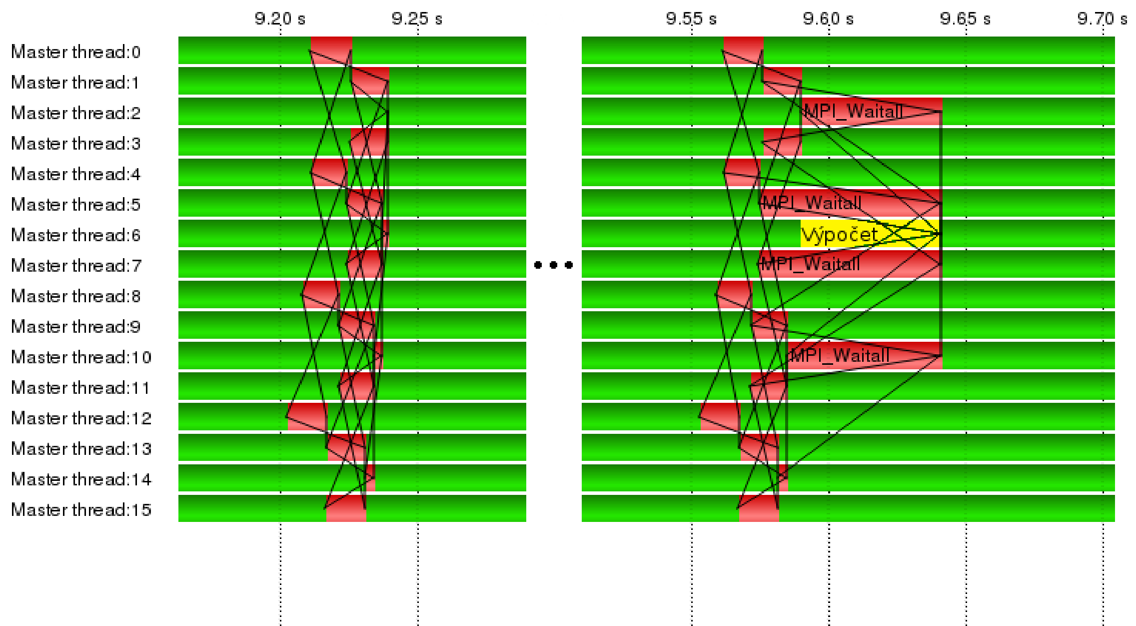
Všechny naměřené hodnoty potvrzují původní předpoklad. Ačkoliv výsledky přesně neodpovídají hodnotě zpomalení, odchylka nepřesahuje 10%.

3.2.3 Vizualizace zpomalení

Zpomalení názorně ilustruje vizualizace průběhu výpočtu získaná pomocí Vampiru [9] na obrázku 3.2. Ten představuje 2 příklady komunikací při výměně halo zón, mezi kterými dojde k zavedení zpomalení. Zelené pruhy značí výpočet a červené komunikaci - konkrétně funkci `MPI_Waitall`. Čarami jsou naznačeny jednotlivé point-to-point komunikace.

Zatímco komunikace vlevo proběhne v horizontu asi 40ms, u komunikace vpravo je patrné čekání v blokující funkci `MPI_Waitall`. Žlutá barva označuje dobu 50ms, kdy uměle zpomalený rank 6 stále počítá, zatímco sousední ranky (2,5,7 a 10) na něj čekají. Jakmile výpočet dokončí, jsou dokončeny i všechny komunikace. Obdobná situace se pak bude opakovat v dalších iteracích.

V konečném důsledku může dojít k rozložení synchronizace v rámci celého cyklu, kdy dochází k překrytí výpočtu a synchronizace v čase. Takový případ demonstruje obrázek 3.3. Jedná se o časový průběh v pokročilé fázi zpožďovaného výpočtu z experimentu na 64 jádrech. Z obrázku je jasně patrné, že jednotlivé komunikace zde probíhají v časové posloupnosti namísto koncentrace do krátkého intervalu. Při takovém výpočtu je značná část výpočetního času věnována pouze čekání na data a synchronizace se rozbíhá napříč iteracemi.



Obrázek 3.2: Vizualizace rozdílu ideální a zpomalené komunikace

Obrázek 3.3: Příklad rozložení synchronizace

Kapitola 4

Návrh řešení

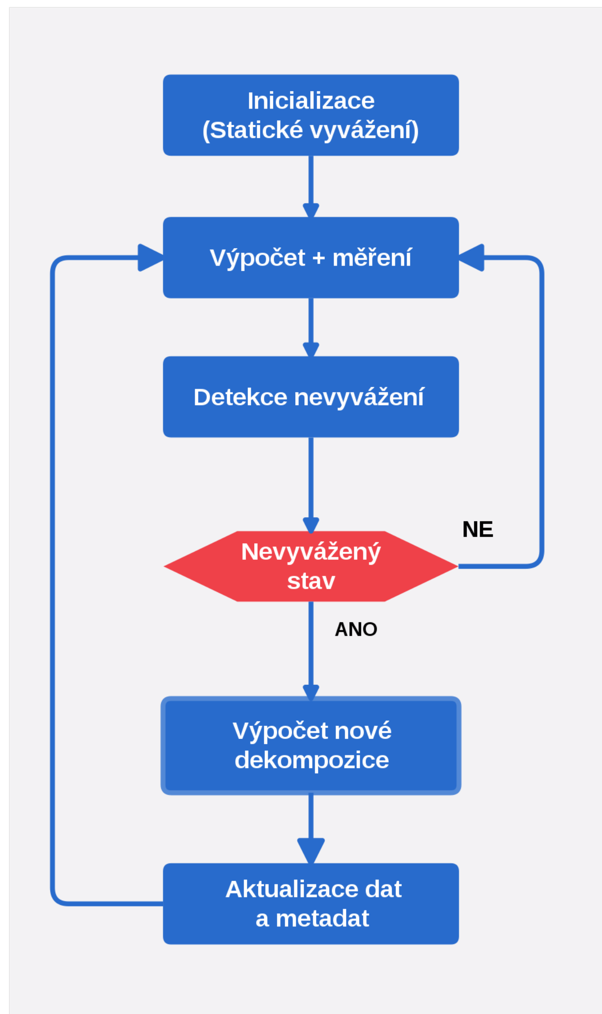
Návrh systému dynamického vyvažování koncepčně vychází z popsané statické varianty, kterou podstatně rozšiřuje. Protože jedním z cílů této práce je vytvořit robustní základ pro další výzkum, byl kladen důraz na genericitu celého řešení, které umožňuje snadnou implementaci řady dalších experimentů s použitím různých nástrojů třetích stran. Systém tedy hierarchicky popisuje jednotlivé dílčí komponenty, nad kterými pracuje. Hlavním cílem je vytvořit systém, schopný dynamického vyvažování při zachování přijatelného poměru mezi genericitou, intuitivním návrhem struktur a efektivitou řešení.

4.1 Vysokoúrovňový popis

V rámci práce byla navržena dynamická 2D dekompozice domény popsané v předchozích kapitolách. Oproti popsanému statickému řešení mohou jednotlivé bloky měnit za běhu téměř libovolně svoji velikost. Tvarově jsou omezeny na obdélník nebo čtverec. Tím se zákonitě mění i počet sousedů a délky sdílených hran mezi nimi, které jsou zásadní pro výměnu halo zón, což je jedním z klíčových problému celého řešení. Dalším problémem je pak měnící se rozdělení dat mezi jednotlivé procesy.

Návrh se snaží o maximální oddělení režie od vypočtu. Samotné simulační API je tvořeno několika jednoduchými voláními, která zajišťují všechny potřebné operace. Lze tak snadno měnit chování modelu bez větších zásahů do kódu simulačního algoritmu, což opět přispívá k větší variabilitě při provádění experimentů.

Jak již bylo uvedeno, proces dynamického vyvažování se typicky skládá ze 3 kroků - měření, detekce a vyvážení. Vyvážení se samozřejmě provádí pouze tehdy, když arbitr (heuristika) uzná, že je to potřeba. Nastavení těchto heuristik je pak pro účinnost celého systému zásadní. Jestliže k vyvažování dojde, nastává přemapování domény mezi jednotlivé procesy a aktualizace potřebných dat a metadat. Rozhodování a přepočet metadat může obecně provádět jeden nebo více procesů. Celý proces vyvažování je ilustrován na obrázku 4.1.



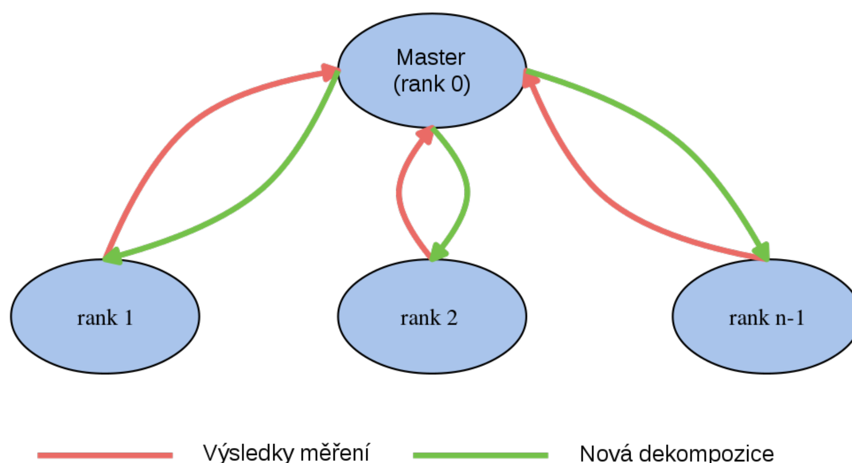
Obrázek 4.1: Proces dynamického vyvažování

V navrženém systému jsou jednotlivé operace rozděleny mezi procesy následovně:

- *Načítání vstupních dat* provádí master proces.
- *Měření* provádí každý proces zvlášť.
- *Rozhodování* provádí pouze master proces.
- *Výpočet nové dekompozice* provádí master proces.
- *Aktualizaci dat* provádí každý proces zvlášť, v součinnosti s ostatními.

4.1.1 Načítání vstupních dat

Podobně jako ve statické verzi, probíhá na začátku každé simulace sekvenční část v podobě načítání dat. Data načte ze souboru master proces a následně distribuuje přidělené bloky ostatním.



Obrázek 4.2: Kooperace procesů při vyvažování

Před samotnou distribucí musí proběhnout počáteční statické vyvážení - tedy iniciální dekompozice domény. Vzhledem k tomu, že model nesbírá žádné informace o stavu systému před spuštěním simulace, je iniciální rozdělení provedeno obdobně jako v případě statické varianty, takže všechny procesy dostanou stejný podíl práce. Při návrhu byla v této fázi zohledněna možnost zajištění lokality procesu při mapování MPI procesů na jednotlivá jádra/uzly clusteru. Vzhledem k tomu, že komunikace mezi procesy v rámci jednoho uzlu (typicky přes sdílenou paměť) je významně rychlejší než mezi několika uzly (přes sběrnici Infiniband), je vzhledem k datovým závislostem výhodnější mapovat sousední bloky na stejný výpočetní uzel. Z tohoto důvodu systém při inicializaci rozlišuje uzel podle *hostname*. Tato funkcionalita však není ve výsledné heuristice pro vyvažování z důvodu nižší priority zahrnuta.

4.1.2 Měření

Každý proces měří svůj výkon sám. V systému figuruje jeden master proces - v tomto případě rank 0, který po určitém počtu iterací sbírá jednotlivá naměřená data od ostatních. Vzhledem k datovým závislostem jsou jednotlivé iterace modelu synchronizovány, takže počet iterací je všude stejný. Data z několika iterací jsou předzpracována heuristikou z důvodu eliminace nárazových zpoždění, která by pominula dříve, než začneme vyvažovat. Cílem je, aby systém nebyl příliš *citlivý* na nevyvážení. Po uplynutí daného počtu iterací je vypočten souhrnný údaj, který je odeslán master procesu. Další výhodou tohoto přístupu je, že není nutné sbírat data v každé iteraci.

4.1.3 Rozhodování

V navrženém systému je rozhodnutí o vyvážení prováděno pouze master procesem, vzhledem k tomu, že má sesbírána naměřená data. Rozhodování, jestli je systém nevyvážený je prováděno jejich srovnáním. Kromě samotných naměřených dat musí do rozhodování vstoupit také informace o aktuální dekompozici.

Ve chvíli, kdy je doména rozdělena nerovnoměrně, má přirozeně každý proces jiný podíl práce a naměřená data jsou úměrná relativní velikosti dlaždice. V dobře vyváženém systému by měly být časové údaje přibližně stejné.

4.1.4 Výpočet nové dekompozice

Nové rozdělení provádí master proces. Jako jediný má kompletní informaci o délce jejich výpočtu a může ji tedy použít k váhování a přidělení nového bloku dat. Toto je jeden z klíčových bodů celého vyvažování. Systém je proto navržen tak, aby mohly být použity různé metody. Musí být pouze schopny generovat kompatibilní popis dekompozice domény, což lze obvykle zajistit patřičným wrapperem.

Vstupem jsou tedy naměřená data sesbíraná od jednotlivých procesů - tedy časy běhu a výstupem je nové rozdělení domény, které by směřovalo k vyváženému stavu. Jakmile je dekompozice vygenerována, může být rozeslána všem procesům, které na jejím základě aktualizují svá metadata.

4.1.5 Aktualizace dat

Ve chvíli, kdy procesy obdrží novou dekompozici, mají kompletní informaci a mohou dopočítat vše potřebné, především aktualizovat metadata a přesunout data modelu. Tím se tento výpočet přirozeně paralelizuje, což urychluje vyvažování. Součinnost jednotlivých procesů je nutná jednak při přesunu dat a jednak při nastavení komunikátorů pro výměnu halo zón. Toto bude podrobně popsáno později.

Mezi nejdůležitější kroky ve fázi aktualizace tedy patří:

- Identifikace sousedních bloků - jejich počet a poloha (navzájem relativní).
- Určení délky sdílených halo zón.
- Identifikace dat, která mají být přesunuta.

Identifikace sousedů a délky halo zón

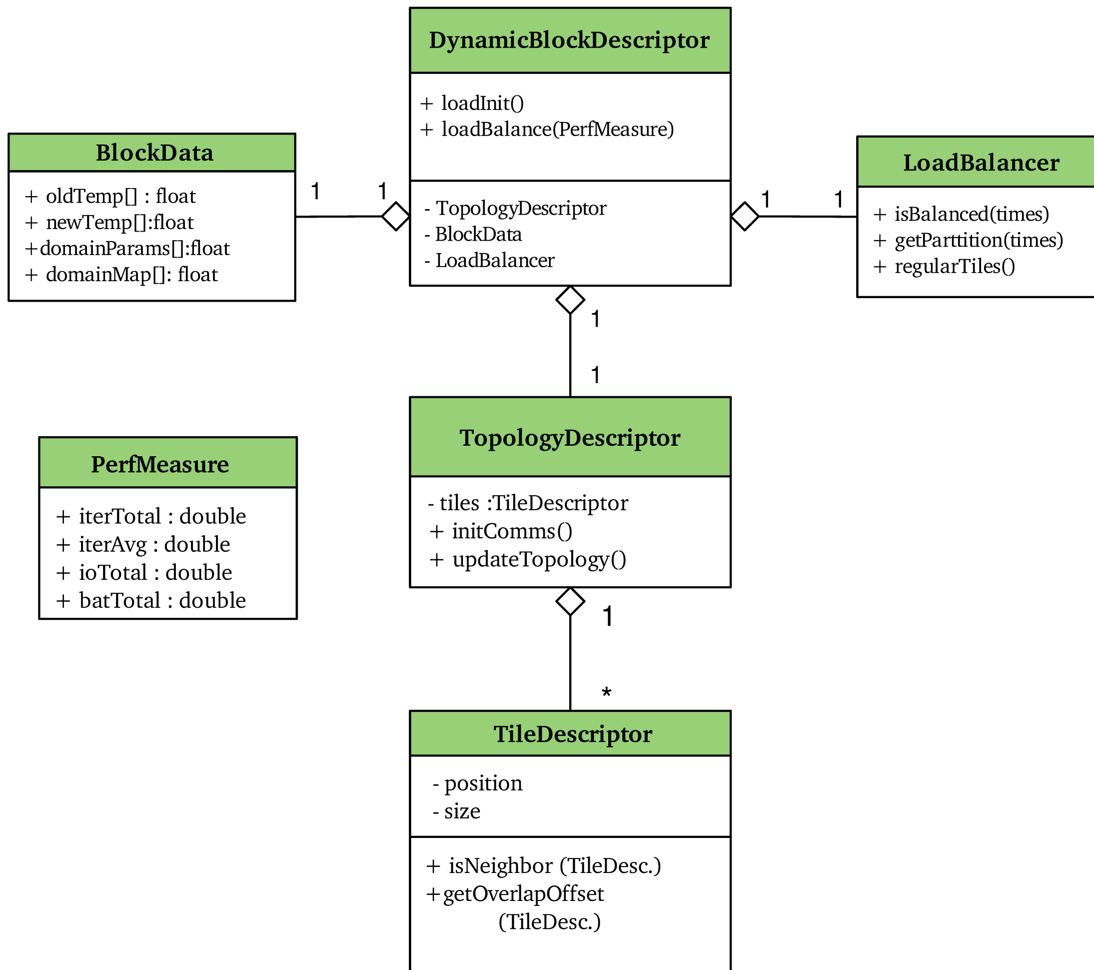
Klíčovou změnou oproti statické variantě modelu je fakt, že každý blok, může mít velmi variabilní počet sousedů. Soused je definován jako libovolný další blok, který s aktuálním blokem sdílí jakoukoli část jakékoliv hrany. Právě zde totiž dochází k výměně halo zón. Ve chvíli, kdy proces zná polohu a velikost všech bloků v doméně, může takový výpočet provést. V této fázi tedy musí každý proces projít všechny bloky v nové dekompozici, identifikovat všechny své sousedy a určit, jakou část halo zón s nimi sdílí.

Identifikace dat k přesunu

Druhým klíčovým krokem je identifikace těch částí domény, které oproti původní dekompozici náleží jinému procesu. V první řadě je tedy potřeba zjistit, kde se tato data nacházejí a pak zajistit jejich přesun. Tato operace je samozřejmě z pohledu každého procesu obousměrná. Data, která náleží procesu X, může vlastnit proces Y a naopak. Každý proces tedy provádí import a export dat. Samozřejmě může nastat i situace, kdy proces nic neimportuje nebo neexportuje. Pokud se však dekompozice změní, jedna z operací je vždy provedena.

4.2 Hierarchie tříd

Vzhledem k tomu, že dynamické vyvažování vyžaduje za běhu udržovat značné množství dat i metadat a provádět s nimi různé výpočty, byla k tomuto účelu navržena hierarchie tříd, ve které každá třída představuje jednu úroveň popisu domény. Klíčové třídy a jejich závislosti jsou zobrazeny na obrázku 4.3.



Obrázek 4.3: Zjednodušený diagram tříd

Třídy v tomto diagramu obsahují jen vybrané atributy, které ilustrují jejich účel. Celý systém dynamického vyvažování je zapouzdřen ve třídě `DynamicBlockDescriptor`. Ta obsahuje 2 klíčové metody `loadInit()` a `loadBalance()`. První z nich se stará o počáteční inicializaci a ta druhá potom o dynamické vyvažování během výpočtu. Tato třída zároveň inicializuje všechny ostatní, jejichž instance obsahuje. Výjimkou je třída PerfMeasure, která existuje samostatně a stará se měření výkonu. Metadata o výpočetní doméně jsou obsažena ve 2 dalších třídách. Třída `TileDescriptor` popisuje jeden blok, především tedy jeho pozici a velikost. Zároveň poskytuje metody pro detekci sousedů a výpočet informací o halo zónách. Celá dekompozice domény je tedy popsána jedním TileDescriptorem pro každý

proces (= blok). Všechny instance pro danou doménu jsou pak obsaženy v atributu třídy `TopologyDescriptor`. Tato třída zároveň zahrnuje veškeré operace prováděné nad doménou jako celkem. Mezi nejdůležitější metody této třídy patří metoda `updateTopology()`, která aktualizuje všechna potřebná metadata.

Třída `LoadBalancer` zapouzdřuje většinu logiky týkající se vyvažování, tj. detekci nevyvážení a výpočet nové dekompozice. V případě dalších experimentů představuje právě tato třída rozhraní pro vyvažování. Může tedy sloužit jako wrapper pro knihovny třetích stran apod. `LoadBalancer` zároveň zahrnuje instanci knihovny `Zoltan`, která je důležitou součástí systému a bude popsána později.

Třída `BlockData` pak obsahuje konkrétní doménová data. Jednak zapouzdřuje data uvnitř DynamicBlockDescriptoru a jednak slouží jako návratová hodnota obou jeho klíčových metod. Poskytuje tedy vždy aktuální data pro simulaci a kromě samotných dat obsahuje také metadata ze tříd napříč celou hierarchií. To k nim umožňuje snazší přístup a zlepšuje čitelnost kódu. Detailnější popis jednotlivých tříd bude uveden později.

4.3 Měření výkonu

Rozhraní pro měření výkonu poskytuje již zmíněná třída PerfMeasure. K měření času využívá volání `MPI_Wtime()` a umožňuje provést měření různých operací zvlášť. Hlavní je měření celkového času iterace, ale zároveň je měřen čas strávený vyvažováním nebo I/O operacemi.

Důležitou součástí je zmíněné předzpracování naměřených hodnot. Pro pokročilejší statistické metody mohou být hodnoty ukládány do historie a zpracovány až po uplynutí patřičné periody, ale v navrženém případě je počítán průměr na což stačí samostatná proměnná. Třída zároveň počítá uplynutí patřičného počtu iterací, kdy má začít detekce.

4.4 Detekce nevyvážení

Detekci nevyvážení provádí metoda `LoadBalancer::isBalanced()`. Navržená metoda je založena na rozdílu mezi nejrychlejším a nejpomalejším časem výpočtu. Vychází z předpokladu, že pokud je doména rozdělena pravidelnou mřížkou, udává nejlepší dosažitelný čas právě nejkratší doba iterace. Pokud je rozdíl mezi nejkratším a nejdelším časem roven určitému násobku nejkratšího času, je výpočet považován za nevyvážený. Tedy:

$$if(max(times) > min(times) \times koef) \rightarrow imbalanced$$

kde *times* jsou naměřené hodnoty a *koef* je násobící koeficient. Ten je pro experimentální účely parametrem simulace, jinak má výchozí hodnotu 1.5.

4.5 Zoltan

Zoltan je knihovna zaměřená na vyvažování zátěže a manipulaci s daty v paralelních systémech [6]. Projekt je vyvíjen v Sandia National Laboratories. Knihovna implementuje řadu algoritmů pro dynamické vyvažování a poskytuje rozhraní pro popis a přesun dat mezi jednotlivými procesy. Jedním z hlavních cílů tohoto projektu je univerzálnost. Je navržena pro použití v různých typech aplikací s rozdílnými typy výpočetních domén. Tento abstraktní přístup s sebou nese i mnohá úskalí a vyžaduje vyšší režii ze strany vývojářů při práci s knihovnou.

Knihovna je implementována v C, byla však doplněna o C++ wrapper s objektovým zapouzdřením. V současné době je dále vyvíjena v rámci projektu Trilinos [11]. Ten sdružuje několik desítek podobných projektů a snaží se je integrovat do sjednocené platformy. Zároveň je zde snaha o podporu novějších standardů C++ s využitím šablon a generik. V rámci projektu Trilinos je vyvíjen Zoltan2 s podporou modernějších přístupů. Podle dokumentace [18] prochází projekt aktivním vývojem a v současné době neobsahuje všechny komponenty.

V rámci této práce byla použita původní verze 1 s C++ wrapperem. Ani ten nepokrývá celou funkcionalitu, takže některé metody je třeba volat s využitím C API, což není problém, protože se jedná opravdu jen o wrapper. Původním záměrem bylo využití jak nástrojů pro vyvažování tak pro migraci dat poskytovaných Zoltanem, ale v případě metod pro vyvažování nastaly problémy právě s popisem domény, ve které je potřeba dodržovat obdélníkový nebo čtvercový tvar jednotlivých bloků. Nakonec jsou tedy využívány pouze nástroje pro datovou migraci.

4.5.1 Metody vyvažování

Zoltan zahrnuje dva základní typy metod pro vyvažování. Prvním z nich jsou metody založené na geometrickém dělení domény a druhým typem jsou metody založené na grafech nebo hypergrafech.

Vzhledem k tomu, že model šíření tepla má přesnou geometrickou doménu a dekompozice v navrženém případě vyžaduje vytvářet obdélníkové bloky, zvolil jsem geometrickou metodu vyvažování. Metody založené na grafech jsou vzhledem k množství publikací v dnešní době používanější, ale vyžadovaly by vytvářet adekvátní reprezentaci modelu. Pro tento případ se jeví geometrická metoda jako přímočařejší řešení. Zvolena tedy byla metoda RCB (Recursive Coordinate Bisection) [2].

Metoda RCB

Jak název napovídá, metoda je založena na rekurzivním dělení souřadnic. V každé iteraci je doména rozdělena na dvě části podél jedné z os. Výběr osy záleží na implementaci. V případě Zoltanu je to osa, ve které je doména delší [1]. U čtvercové domény je tedy výběr nedefinovaný. Ve výchozím stavu je prostor dělen na poloviny. Velikost jednotlivých bloků lze ovlivnit váhováním, čímž se generuje patřičně vyvážené rozdělení.

Zoltan bohužel umožňuje popsat výpočetní doménu pouze pomocí souřadnic jednotlivých objektů. Velikost ani hranice samotné domény specifikovat nelze a nelze ani blíže definovat omezení pro tvar jednotlivých bloků. Přestože by metoda měla přirozeně generovat obdélníkové nebo čtvercové bloky, při experimentech se nepodařilo tuto podmínku dodržet. V případě rovnoměrného rozložení bloků fungovala metoda korektně, avšak při experimentech s váhováním se ukázalo, že generuje velmi nepravidelné obrazce. Bohužel se nepodařilo nastavit patřičné omezení.

Použitá metoda

Po neúspěchu s implementací metody RCB, byla v rámci této práce navržena vlastní metoda. Ta je založena na kombinaci 1D vyvažování ve 2D dekompozici. Její princip spočívá v tom, že vyvažuje každý řádek 2D dekompozice zvlášť. Ubírá tedy jednotlivým blokům jeden stupeň volnosti. Na druhou stranu využívá potenciál navrženého systému, protože generuje různě se překrývající bloky s různým počtem sousedů.

[1] http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_rcb.html

4.5.2 Zoltan API

Základní prostředkem pro interakci Zoltanu s uživatelským modelem je sada definovaných callback funkcí. Každý nástroj v Zoltanu vyžaduje implementovat určitou podmnožinu z celé řady těchto funkcí.

Zoltan popisuje data výpočetní domény jako *objekty*. Tyto nemají nic společného s objektově orientovaným paradigmatem programování. Jedná se o uživatelem definované části výpočetní domény, ať už jsou jakékoliv, které mají přiděleno unikátní ID. Objekty mohou být různě velké i komplexní v závislosti na požadovaných vlastnostech. Zoltan pak například při migraci dat volá patřičné callbacky s ID objektu a je na uživateli jak jej zpracuje. Callbacků jsou řádově desítky [2].

Všechny callbacky se registrují při inicializaci knihovny. V návrhu této práce jsou realizovány jako statické metody DynamicBlockDescriptoru. Ten je zároveň předáván pomocí ukazatele *this* jako parametr callbacku, což všechny callbacky umožňují. Mohou tak přistupovat podle potřeby kde všem datům a metadatům modelu.

[2] http://www.cs.sandia.gov/Zoltan/ug_html/ug_index.html - callbacky jsou v odkazovaném seznamu velkými písmeny, ostatní jsou uživatelské funkce

Kapitola 5

Implementace

Cílem této kapitoly je hlubší popis modelu, implementovaného na základě prezentovaného návrhu. Budou zde podrobněji rozebrána řešení, použitá pro dekompozici domény, migraci dat a rozhraní mezi backendem pro vyvažování zátěže a simulačním algoritmem.

5.1 Dekompozice domény

Jak již bylo řečeno, model pracuje se čtvercovou doménou o délce hrany, která je rovna mocninám dvou. V případě statické implementace je doména rozdělena na bloky stejné velikosti. V závislosti na počtu procesů, mezi které doménu rozdělujeme jsou bloky čtvercové nebo obdélníkové (počet procesů je roven sudé nebo liché mocnině dvou).

V případě dynamického vyvažování je však potřeba rozdělit doménu tak, aby bylo možné přidělovat části domény po menších blocích a v případě potřeby je mezi procesy přesouvat. Doména je tedy rozdělena hned ve dvou úrovních. Jednu úroveň představují jednotlivé bloky podobně jako ve statické variantě a druhou, nižší úroveň, představují části domény, se kterými pracujeme jako s elementární jednotkou dat. V navrženém modelu tyto části odpovídají Zoltan objektům, o kterých byla zmínka v předchozí kapitole. Velikost objektu, který představuje elementární množství doménových dat, má zásadní vliv na efektivitu vyvažování. Je proto nutné najít kompromis mezi množstvím objektů, které je možno přemapovat mezi procesy a množstvím režie, nutné k práci s nimi. S tím souvisí i efektivita jejich přenosu.

V rámci práce byly objekty navrženy – podobně jako samotné bloky – jako čtvercové submatice výpočetní domény. Na velikosti těchto objektů závisí jak rychlosti přemapování domény tak variabilita vyvažování. Velikost je proto jedním z parametrů simulace.

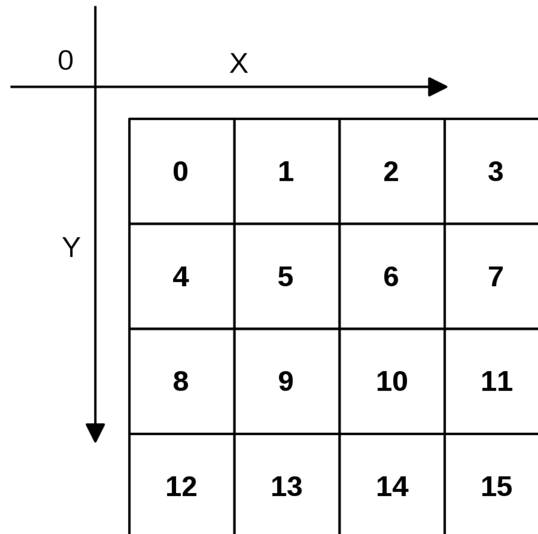
Vliv na rychlost vychází z faktu, že Zoltan přistupuje ke každému objektu zvlášť. Pro hromadný přenos objektů poskytuje Zoltan samostatné callbacky. Ty ovšem nebyly implementovány. V každém případě jsou pro každý objekt volány callbacky `pack` a `unpack` [1], které zajišťují přesun objektu do bufferu a zpět.

Vliv na variabilitu je pak zřejmý. Objekt je nejmenší subdoména, kterou lze přidělit některému bloku. Pokud bychom tedy pracovali s objekty o velikosti 1, přesouváme každý bod domény zvlášť a variabilita by tak byla maximální. Naopak pokud by byly objekty stejně velké jako bloky, vyvažování nebude prakticky možné.

Doména je tedy rozdělena na jednotlivé objekty tak, že jsou číslovány z levého horního rohu. Číslo odpovídá globálnímu ID (`GID`), které používá Zoltan. `GID` jednoznačně iden-

[1] Přesněji ZOLTAN_PACK_OBJ_FN a ZOLTAN_UNPACK_OBJ_FN

tifikuje objekt, podobně jako ve statické variantě rank jednoznačně identifikoval blok. Na základě `GID` tedy lze určit polohu objektu a naopak, což je klíčové pro jeho přesun mezi procesy. Rozdělení domény ilustruje obrázek 5.1.



Obrázek 5.1: Rozdělení domény na objekty

Tento přístup s sebou přináší ještě jednu důležitou vlastnost. `GID` nahrazuje rank při určování pozice. Poloha a velikost bloku je implementačně dána množinou objektů, které jsou tomuto bloku přiděleny. Tato množina musí samozřejmě představovat obdélník nebo čtverec, jelikož jiné tvary jsou nepřípustné. Důležité je však to, že rank procesu je nezávislý na poloze a velikost bloku, který mu náleží.

V rámci topologie jsou přesto popsány jednotlivé bloky - jejich pozice a velikost - třídou `TileDescriptor`. Při každé změně topologie jsou tyto informace odvozeny z množiny přiřazených bloků. Objekty mají po celou dobu běhu fixní velikost, takže o nich není nutné udržovat žádnou další informaci.

5.2 Komunikace sousedních bloků

Dalším klíčovým prvkem je zajištění komunikace mezi sousedy a výměna jejich halo zón. Jsou-li všechny objekty přiřazeny některému procesu, vzniká dekompozice s množinou navazujících bloků, které tvoří sousedy. Mezi nimi je třeba efektivně komunikovat. Abychom celý proces zjednodušili je komunikace řešena pomocí izolovaných komunikátorů.

5.2.1 MPI komunikátor

Komunikátor v MPI představuje kontext, zapouzdřující skupinu procesů, které mezi sebou komunikují. Výchozím komunikátorem je vždy `MPI_COMM_WORLD`, z něhož vznikají všechny ostatní. Každému komunikátoru navíc naleží skupina (`MPI_Group`), která představuje onu množinu procesů patřících do komunikátoru.

Důležitým detailem je, že každý komunikátor má svoje vlastní číslování procesů. Komunikátor lze vytvořit několika způsoby. Nejčastějším z nich je kolektivní volání

`MPI_Comm_split`. Tato funkce rozdělí daný zdrojový komunikátor (například `MPI_COMM_WORLD`) na N nových komunikátorů. Číslo N je závislé na počtu barev - tedy různých čísel, které jednotlivé procesy předají jako parametr. Dalším důležitým parametrem je klíč, který udává, jakým způsobem budou číslovány ranky. Čím menší klíč proces předá, tím nižší rank v novém komunikátoru dostane.

Pokud například chceme rozdělit všechny procesy do dvou komunikátorů na sudé a liché, sudé ranky předají barvu například 0 a liché 1. Pokud jako klíč použijí svůj rank, vzniknou 2 komunikátory. V prvním bude rank 0 stále rank 0, ve druhém bude rank 1 nově jako rank 0. Všechny vyšší ranky, dostanou odpovídající vyšší číslo.

5.2.2 MPI skupiny

Abychom byli schopni určit mapování ranků mezi jednotlivými komunikátory, je třeba použít zmíněné skupiny. Pokud ze dvou komunikátorů získáme jim náležící skupinu `MPI_Group`, můžeme voláním `MPI_Group_translate_ranks` získat ranky náležící jednomu procesu v různých komunikátorech. Tato funkcionalita je nutná k tomu, abychom například byli schopni určit proces v roli roota v novém komunikátoru, v situaci, kdy známe jeho rank pouze ve výchozí komunikátoru.

5.2.3 Princip komunikace

Výše uvedené koncepty jsou klíčové pro komunikaci mezi sousedy. V dynamickém systému s 2D dekompozicí řešíme 2 základní problémy. Jedním je variabilní počet sousedů a druhým je různá délka halo zón, která z toho vyplývá. Pracovat v takovém případě například s datovými typy, které by popisovaly jednotlivé úseky, by bylo velmi nepraktické. Byl proto zvolen přístup distribuce halo zón pomocí kolektivní neblokující komunikace `Iscatterv`. Ta umožňuje distribuovat celou halo zónu mezi různý počet sousedů jedním voláním. Je však velmi vhodné, aby sousedi tvořili samostatný komunikátor.

V tomto systému tedy každý blok definuje vlastní komunikátor, do něhož spadají všichni jeho sousedi. Postup jejich vytváření popisuje pseudokód 5.1.

Listing 5.1: Pseudokód tvorby komunikátorů

```
1  for ( i in ranks(MPI_COMM_WORLD))
2  {
3          if( i == myRank){
4
5                  MPI_Comm_split(MPI_COMM_WORLD , color1 , i , newComm)
6          }else if(i is in  myNeighbors[]){
7
8                  MPI_Comm_split(MPI_COMM_WORLD , color1 , i , newComm)
9          }else{
10                 MPI_Comm_split(MPI_COMM_WORLD , MPI_UNDEFINED , i , newComm)
11         }
12 }
```

Jak je z předchozího kódu patrné, všechny procesy synchronně iterují přes jednotlivé ranky. Jakmile přijde proces na řadu, vytváří komunikátor, ve kterém bude rozesílat halo zóny. Do tohoto komunikátoru se zároveň zařadí všechny ranky, které sousedí s rankem `i`. Ostatní ranky žádný komunikátor nevytváří, takže předávají barvu MPI_UNDEFINED.

Určení root ranku

V každém nově vytvořeném komunikátoru je potřeba určit rank root komunikátoru. Jednotlivé procesy používají jako klíč komunikátoru svůj rank a zároveň neznají ranky všech sousedů ranku i, které do komunikátoru vstupují. Root tedy může mít v novém komunikátoru libovolný rank.

Ve skutečnosti každý proces zná celkovou topologii, takže by všechny ranky vstupující do komunikátoru dopočítat mohl, ale s výhodou lze využít právě překladu poskytovaného MPI. Všichni sousedi ranku `i` tedy obdrží jeho ekvivalent v novém komunikátoru. Ten se pak stane rootem při volání `Iscatterv`.

Rank `i`, který bude zdrojový naopak všechny svoje sousedy zná, což je nutné pro výpočet parametrů `counts` a `displacements`, tedy offsetu ve zdrojovém bufferu a počtu prvků, které mají být rozeslány.

5.2.4 Halo zóny

V tuto chvíli má na základě algoritmu 5.1 každý proces vytvořen jeden komunikátor, ve kterém je root a zároveň do něj patří všichni jeho sousedi.

Abychom mohli rozeslat halo zóny pomocí `Iscatterv`, potřebujeme je nejprve serializovat do bufferu a určit, kterou část má obdržet který soused. Toto zajišťuje metoda `TileDescriptor::getOverlapOffset` [2]. Vstupním parametrem je sousední blok a návratovými hodnotami jsou právě offset a počet sdílených prvků. Výpočet je proveden na základě souřadnic a velikosti obou bloků. Aby bylo možné poslat celou halo zónu jedním voláním, je třeba uložit prvky do paměti tak, aby je bylo možné v libovolném místě rozdělit.

Halo zóny na horním a dolním okraji se ukládají po sloupcích, kdežto na levém a pravém okraji se ukládají po řádcích. Navíc, aby bylo možné úseky správně indexovat, ukládá se do bufferu nejprve horní a pravý okraj a poté levý a dolní okraj. Abychom se v implementaci vyhnuli neefektivnímu čtení po sloupcích, jsou data čtena po řádcích a do bufferu se zapisují nejprve na sudé a poté na liché indexy.

K offsetu z levého a dolního okraje se přičte délka horního a pravého okraje. Tím je zajištěno, že každý sousední blok dostane data na správném offsetu ve správném pořadí.

Indexování halo zón a rozložení dat v bufferu ilustruje obrázek 5.2.

Při volání `Iscatterv` je předáván jak offset, tak množství dat, které má proces dostat. Tato data jsou seřazena podle ranků v daném komunikátoru. Rank, který data odesílá v tomto případě žádná dostávat nemá (oproti běžnému chování `Scatter`). Na odpovídající pozici proto předává 0.

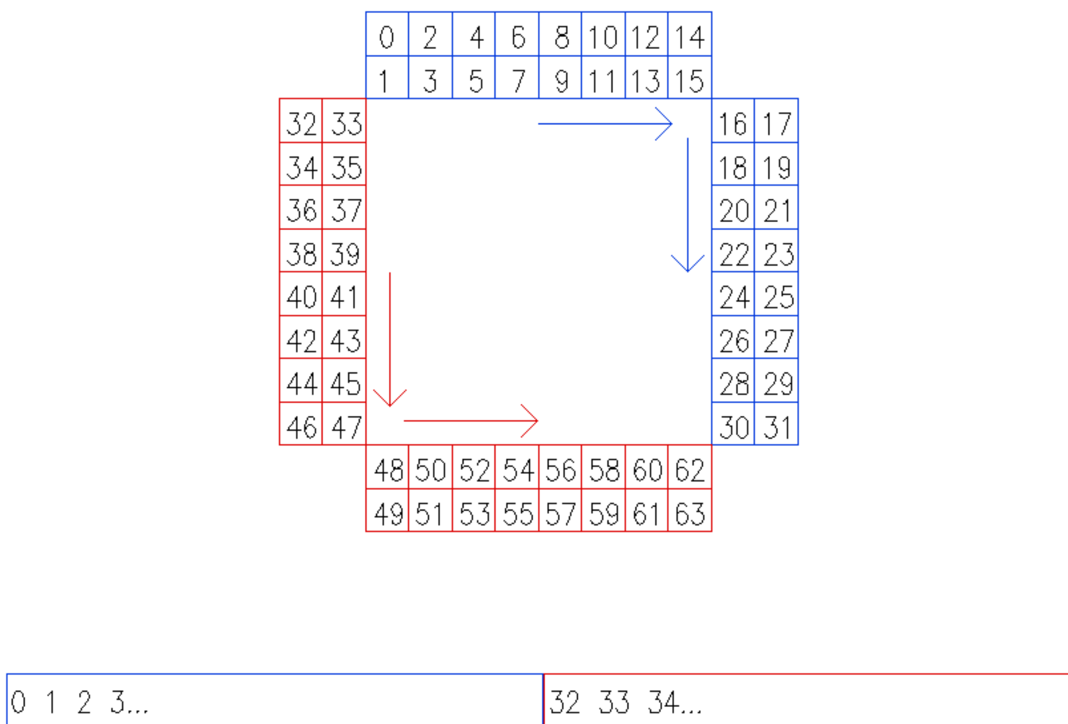
5.3 Migrace dat mezi procesy

Další klíčovou komponentou systému dynamického vyvažování je přesun dat mezi procesy. Jak bylo popsáno v úvodu, toto je úzce spjato s knihovnou Zoltan a popisem dat pomocí objektů, jakožto elementární popis dat.

5.3.1 Integrace knihovny Zoltan

Zoltan je v systému integrován na dvou místech. Samotná instance se nachází ve třídě `LoadBalancer` vzhledem k původnímu záměru využívat i funkcionalitu pro vyvažování

[2]prototyp `unsigned getOverlapOffset(const TileDescriptor & tile, unsigned & cnt) const`



Obrázek 5.2: Ilustrace výpočtu halo offsetu

zátěže. Ve výsledné implementaci se používá pouze pro migraci dat. Jak bylo popsáno výše, základem rozhraní Zoltanu je řada callback funkcí. Tyto funkce jsou implementovány ve třídě `DynamicBlockDescriptor`. Mezi klíčové callbacky patří:

- ZOLTAN_OBJ_SIZE_FN - vrací velikost jednoho objektu v bytech
- ZOLTAN_PACK_OBJ_FN - serializuje objekt do bufferu
- ZOLTAN_UNPACK_OBJ_FN - deserializuje objekt zpět do domény

V systému je ještě implementována řada dalších callbacků, které byly určeny pro vyvažování.

Přesun objektů zajišťuje funkce `Migrate`[3]. Ta má celou řadu parametrů, které v zásadě popisují množiny objektů pro import/export z/do aktuálního procesu. Množiny jsou implementovány pomocí pole GID (případně lokálních ID, ty jsou v projektu totožné s GID). Ke každé položce v seznamu pro import/export ještě náleží zdrojový/cílový rank. Celkově má funkce migrate 10 parametrů. Kromě čísla procesu, které odpovídá MPI ranku, ještě Zoltan umožňuje rozlišovat na části (parts). Ty se v projektu opět nepoužívají a jsou vždy nastaveny na 0.

Pokud tedy má dojít k nějakému přesunu, je třeba aby každý rank inicializoval patřičná pole správnými hodnotami. Samotný přesun se pak realizuje kolektivním voláním migrate. Během tohoto volání dochází postupně k volání callbacků, dokud nejsou všechny požadované objekty přemístěny.

[3]http://www.cs.sandia.gov/Zoltan/ug_html/ug_interface_mig.html#Zoltan_Migrate

Při implementaci se ještě objevil jeden důležitý detail. Zoltan sám o sobě komunikuje a vyžaduje přidělit MPI komunikátor. Výchozím je `MPI_COMM_WORLD`. V souvislosti s vytvářením mnoha komunikátorů pro přemísťování halo zón se použití `MPI_COMM_WORLD` pro Zoltan ukázalo jako problémové a docházelo k chybám a to přesto, že by veškeré akce Zoltanu měly být řízeny voláním uživatelských funkcí. Při inicializaci bylo třeba vytvořit duplicitní komunikátor k výchozímu, aby se oddělil komunikační kontext.

Konfigurace Zoltanu je prováděna v metodě `DynamicBlockDescriptor::zoltanInit()`. Zde je především předáván samotný volající objekt, jako parametr callbacků představující uživatelská data.

5.3.2 Rozhraní Zoltanu pro migraci dat

Pro migraci dat poskytuje Zoltan zmíněné callbacky pack a unpack, které opět mají celou řadu parametrů. Klíčové jsou 3. Buffer pro data, ukazatel na uživatelská data a číslo objektu (případně 4., chybová proměnná).

Při přesunu objektu jsou postupně jednotlivé callbacky volány. Nejprve je volán callback pro velikost objektu při alokaci patřičného bufferu a poté může začít samotný přesun. Vzhledem k tomu, že model je popsán pomocí 3 polí - teplota, koeficient vodivosti a mapa domény, jsou při migraci objektu přenášeny ve skutečnosti data z těchto tří polí.

Z důvodu efektivity jsou ukládány do jednoho bufferu v pořadí - matice teplot, parametry, mapa domény. Nepříjemnou komplikací je rozdílnost typů (float, int), kterou je třeba brát v úvahu při práci s ukazateli. Jednotlivé objekty jsou kopírovány pomocí `std::memcpy`.

V systému existují dva rozdílné případy pro migraci dat. Prvním z nich je běžná distribuce a redistribuce dat a druhým z nich je sekvenční I/O. Pro tyto účely mají i callbacky dva různé módy chování. V případě (re)distribuce jsou přesouvány obsahy všech 3 zmíněných polí, v případě sériového I/O pouze pole teplot.

5.3.3 Zdrojová a cílová pole

Při přesunu objektů je nutné brát v úvahu, že zdrojový a cílový blok mohou mít zcela jinou pozici i velikost. Před samotnou migrací je tedy nutné připravit pole pro nový blok. Pro tyto účely jsou ve třídě `DynamicBlockDescriptor` dva atributy. Aktuální data, nad kterými se provádí výpočet jsou v objektu `BlockData`, který obsahuje řadu dalších atributů a pole, alokovaná pro nový blok, jsou v objektu `TempBlock`. Celkový postup přesunu objektů sestává z těchto kroků:

1. Na základě nové dekompozice určíme velikost a pozici nového bloku.
2. Alokujeme patřičná pole v objektu `TempBlock`.
3. Nastavíme seznamy pro import/export.
4. Voláme funkci Migrate.
5. Přesuneme objekty, které zůstávají přiděleny stejnému bloku.

Vzhledem k tomu, že fixní je pozice objektů a pozice bloků se mění, jsou objekty zapisovány na základě relativní pozice v rámci bloku. Metoda pack vždy čte z aktuálních dat, kdežto metoda unpack vždy zapisuje do pole pro nový blok (s výjimkou případu s I/O). Jakmile jsou objekty Zoltanem přeneseny, volá se metoda `movePersistObj`, která jednak přesune lokální objekty a jednak aktualizuje pole. Od této chvíle je migrace dokončena.

5.3.4 Výpočet seznamů pro migraci

Protože byl v rámci práce implementován vlastní algoritmus pro vyvažování, bylo nutné vyřešit generování seznamů objektů, které mají být Zoltanem přesunuty. Ve chvíli kdy je vypočtena nová dekompozice, můžeme porovnat původní a novou a na jejím základě seznamy vytvořit. Volání funkce Migrate je kolektivní, ale každý proces mu předává vlastní seznamy objektů. Ty musí být předem vypočteny pomocí množinových operací, porovnáním staré a nové dekompozice. Tento problém řeší metoda `resolveMigration`. Nejprve jsou vypočteny seznamy objektů náležící jednotlivým blokům ve staré i nové dekompozici. Protože kromě samotných GID objektů je nutné také určit zdrojové/cílové ranky, jsou seznamy uloženy v kontejneru `std::map`.

Nejprve jsou pomocí množinových operací vypočteny množiny Import, Export a Persist, a to následovně:

$$Import = NewGIDs \setminus OldGIDs$$
$$Export = OldGIDs \setminus NewGIDs$$
$$Persist = NewGIDs \cap OldGIDs$$

kde $NewGIDs$ je množina všech `GID` nově přiřazených objektů a $OldGIDs$ je množina `GID` přiřazených v původní dekompozici.

Posledním krokem je lokalizace objektů mezi jednotlivými procesy, aby mohly být nastaveny zdrojové/cílové ranky. Ta je realizována průchodem výše popsané datové struktury.

5.4 Dynamické vyvažování

V tuto chvíli byly popsány všechny komponenty umožňující pracovat s dynamickou 2D dekompozicí domény. Dalším krokem je implementace samotného dynamického vyvažování. Celý vyvažovací proces byl stručně popsán již v kapitole věnované návrhu, kde je zároveň stručný úvod do použité vyvažovací metody. Implementovaná metoda je o něco jednodušší než původně navrhovaná metoda RCB. Zároveň je oproti ní o něco méně efektivní kvůli omezení v jednom rozměru. Stále ale umožňuje využít potenciál 2D dekompozice.

Algoritmy dynamického vyvažování jsou implementovány ve třídě `LoadBalancer`. Základ tvoří metoda `splitByPerform`.

5.4.1 Proporcionální dělení

Metoda `splitByPerform` zajišťuje řádkově orientované rozdělení zátěže. Vstupem metody jsou naměřené časové údaje a výstupem jsou počty objektů, přidělené jednotlivým rankům na řádku.

Cílem metody je přidělit objekty v řádku tak, aby bloky s nejdelším časem dostaly nejmenší podíl práce a naopak. Objekty jsou v každém řádku přidělovány po celých sloupcích. Výška řádku je po dobu simulace fixní.

K tomu poslouží inverze časových hodnot, tedy

$$t_i^{-1}, \forall t_i \in times \tag{5.1}$$

kde $times$ je vektor naměřených časů. Dále je vypočten jednotkový počet přidělovaných sloupců jako

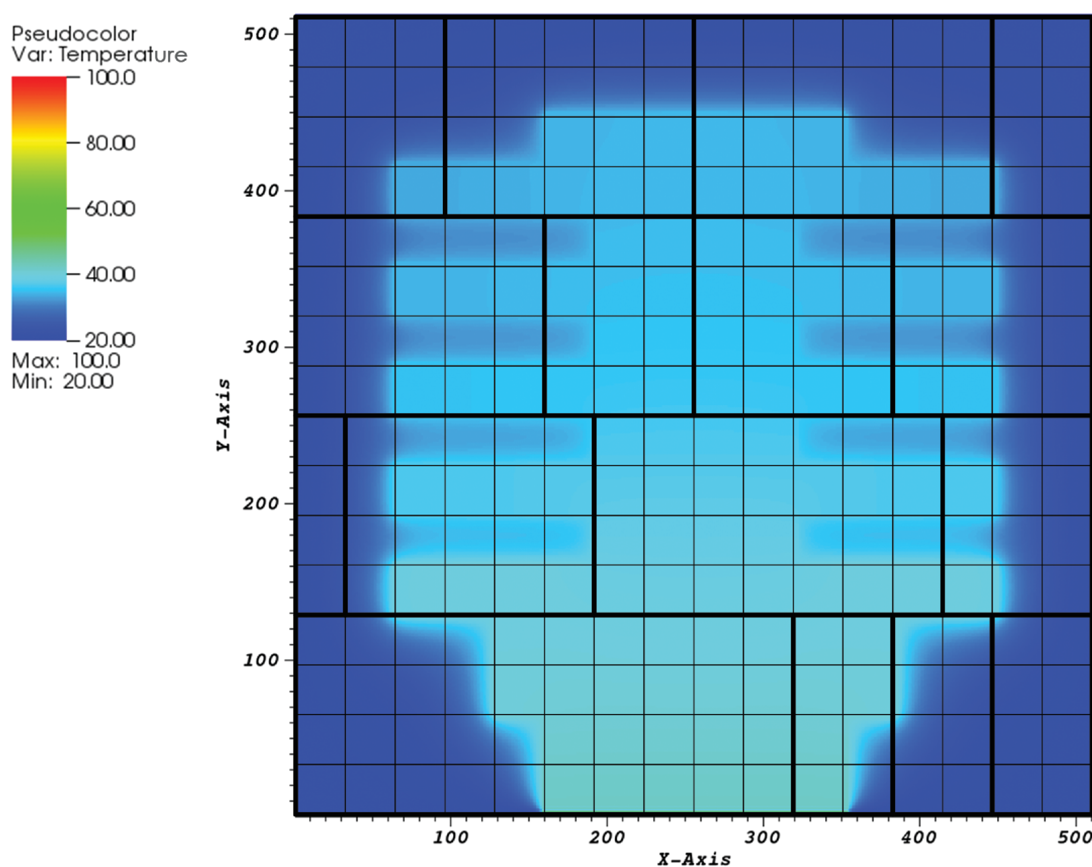
$$poc_objektu/sum(times) \tag{5.2}$$

a touto jednotkou jsou vynásobeny jednotlivé invertované hodnoty. Ty jsou nakonec zaokrouhleny na celá čísla a zbylé objekty (vlivem zaokrouhlení) jsou rovnoměrně rozděleny mezi bloky. Vzniká tak proporční rozdělení objektů mezi procesy.

Pokud dojde vlivem zaokrouhlování k tomu, že nejsou přiděleny všechny dostupné objekty, nebo je jich naopak přiděleno více, jsou objekty přidány nebo odebrány blokům iterativně po jednom, dokud není tato nerovnováha napravena. Tím vzniká relativně rovnoměrné rozložení. Pokud je rozdíl malý oproti počtu bloků, týká se pouze prvních několika bloků v řádku.

Dalším důležitým detailem je, že systém nebyl navržen tak aby blokům odebral práci úplně, takže minimální počet je 1, tedy 1 sloupec přidělených objektů.

Pomocí této metody je rozdělen každý řádek bloků v případě nevyváženého stavu. Metoda `getPartition()` pak generuje novou topologii. Příklad dekompozice domény po vyvážení ilustruje obrázek 5.3.



Obrázek 5.3: Ilustrace domény po vyvážení

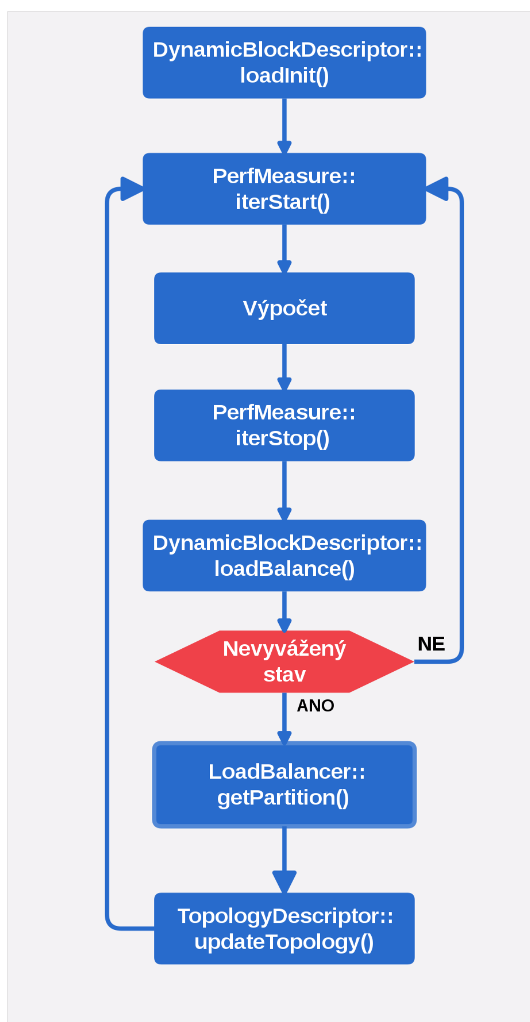
Obrázek ilustruje situaci, kdy na doméně 512^2 s objekty o velikosti 32^2 nastane nevyvážení. Tenké čáry představují objekty, tlusté čáry pak jednotlivé bloky. Počet objektů přidělených blokům v jednotlivých řádcích proporčně nepřímo úměrně odpovídá naměřeným časovým hodnotám.

5.4.2 Obnova rovnoměrného rozdělení

Po dobu trvání příčiny zpomalení lze předpokládat, že vzhledem k vyvážení budou měřené hodnoty přibližně stejné v závislosti na tom, jak dobře se povede práci přerozdělit. Jakmile však příčina zpomalování pomine, hodnoty se začnou lišit a budou přímo úměrné nerovnoměrnému rozdělení práce. Takovou situace je opět třeba detekovat a obnovit iniciální rozdělení, případně vyvážit nově vzniklý nevyvážený stav.

5.5 Rozhraní pro simulaci

Cílem této části je popsat rozhraní mezi backendem systému pro vyvažování a simulačním algoritmem. Rozhraní bylo navrženo s ohledem na jednoduchost. Většina operací spojená s vyvažováním probíhá na pozadí a samotné vyvažování nemusí být využito vůbec. Celý proces je zobrazen v diagramu 5.4. Jednotlivá volání přibližně korespondují s diagramem 4.1.



Obrázek 5.4: Implementace vyvažování, diagram aktivit

Základ rozhraní tvoří třída `DynamicBlockDescriptor`. Ta musí být před simulací inicializována. Jak už bylo popsáno v návrhu, třída vytváří instance ostatních tříd. Poskytuje 2 základní metody - `loadInit()` a `loadBalance()`.

Metoda `loadInit()` řídí proces inicializace a distribuce dat. Musí být volána před začátkem simulace. Metoda realizuje statické rovnoměrné rozdělení práce mezi procesy. K distribuci se opět využívá mechanismů poskytovaných knihovnou Zoltan. Dojde tedy k rozeslání dat formou objektů z ranku 0, který je načítá ze vstupu mezi ostatní procesy. Metoda vrací objekt `BlockData`, který představuje data, nad kterými je prováděn výpočet spolu s metadaty jako jsou velikost bloku a informace pro MPI komunikace.

Druhou klíčovou metodou je `loadBalance()`. Ta již realizuje dynamické vyvažování nad inicializovaných modelem. Chování metody kopíruje proces vyvažování, ilustrovaný schématem 4.1. Nejprve jsou sesbírána naměřená data pomocí kterých metoda `isBalanced()` detekuje případné nevyvážení. V případě, že je detekováno, jsou tyto hodnoty použity pro výpočet nové dekompozice metodou `LoadBalancer::getPartition()`. Nová dekompozice je pak distribuována všem procesům, které ji použijí pro výpočet množin objektů, určených k migraci metodou `resolveMigration`. Následně je migrace provedena a posledním zásadním krokem je volání metody `TopologyDescriptor::updateTopology`, která aktualizuje veškerá metadata. Metoda `loadBalance()` opět vrací objekt `BlockData` s aktuálními daty pro pokračování simulace.

Pro účely sériového I/O a sesbírání dat do ranku 0, který data zapisuje, slouží metoda `collectData()`.

Další pomocné funkce

Během simulace je použito několik dalších funkcí pro lepší čitelnost a udržovatelnost kódu. K dispozici jsou 2 šablonové funkce pro přesun halo zón z/do bufferu. Samotné buffery jsou kvůli změnám velikosti zapouzdřeny ve třídě `HaloBuffers`. Dále je zde funkce `ExchangeHaloZones()`, která se stará o iniciální výměnu halo zón jak pro pole teplot, tak pole parametrů. Během výpočtu se vyměňuje pouze pole teplot.

Pro ošetření chyb při MPI komunikacích je k dispozici inline funkce `MPI_assert()`. Ta v případě nenulové návratové hodnoty vrací výjimku spolu s předaným textem. Volitelným parametrem je makro LOCATION, které využívá standardní makra ___FUNC___, ___FILE___ a ___LINE___. V případě překladače, který tato makra nepodporuje lze LOCATION předefinovat v *Asserts.h*.

5.5.1 Validita modelu

Validita navrženého paralelního řešení byla ověřována vůči sekvenční verzi porovnáním výsledných matic teplot na úroveň zaokrouhlovacích chyb aritmetiky v plovoucí desetinné čárce typu `float`. Tolerance byla nastavena na 1×10^{-3}. Zároveň byly průběžně zaznamenány průměrné teploty v prostředním sloupci. Dalším rámcovým ověřením je vizualizace pomocí nástroje *VisIt* [12].

5.6 Vývoj a dokumentace

Pro vývoj projektu byl použit verzovací systém Git s frontend nástavbou GitLab [8]. Ten kromě samotného verzování umožňuje průběžné dokumentování pomocí *issues* a jejich párování s odpovídajícími větvemi. S pomocí těchto nástrojů byly vytvářeny větve

pro každou zásadní přidávanou funkcionalitu. Po otestování byly větve spojeny (merge) s větví master. Pomocí issues jsou popsány zásadní koncepty a jejich modifikace, ke kterým v průběhu vývoje docházelo. Zdrojový kód byl průběžně komentován. Třídy a metody jsou komentovány kompatibilně s doxygenem.

5.6.1 Veřejný repozitář

Celá práce včetně dokumentace a zdrojových kódů je dostupná na GitHubu v repozitáři `https://github.com/samit3n/Dynamic-Load-Balancing`. Kompletní export původního pracovního repozitáře z GitLabu je uložen na CD přiloženém k této práci, dostupný v archivu FIT VUT Brno.

5.7 Kompilace a spuštění

Model je implementován v jazyce C++ ve verzi ISO C++11. Pro kompilaci je k dispozici Makefile ve složce *Sources*. V rámci experimentů byl převážně používán překladač Intel, resp. wrapper mpicxx.
Konkrétně byly použity tyto moduly:

- `ml intel/2017.00`
- `HDF5/1.8.16-intel-2017.00`
- `Zoltan`

Na Anselmu lze použít Zoltan z instalovaného modulu `trilinos/11.2.3-icc`. Na Salomonu lze použít binárku, přiloženou na CD, nebo jej zkompilovat podle návodu v příloze A.

Kapitola 6

Experimenty

Tato kapitola se věnuje experimentům s implementovaným modelem. Experimenty jsou zaměřeny jednak na otestování dynamického vyvažování zátěže na různých scénářích a jednak na ověření výkonu vrstvy pro migraci objektů, který je pro efektivitu vyvažování zásadní. První experiment byl prováděn na obou clusterech, další potom pouze na Anselmu.

6.1 Migrace dat

Jednou z důležitých vlastností modelu je popisovaný způsob migrace dat. Vzhledem k tomu, že data jsou přesouvána mezi procesy při každém vyvažování, je efektivita této operace pro výkon zásadní. Data jsou migrována jako jednotlivé objekty, tedy sub-matice celé domény, které mají variabilní velikost. Cílem tohoto experimentu je ověřit vliv velikosti objektu na rychlost migrace.

Vycházíme z předpokladu, že efektivita vyvažování je nepřímo úměrná velikosti objektu. Tedy, že čím menší objekt, tím větší variabilita velikosti bloků a tím více možností vyvažování. Zároveň lze předpokládat, že s rostoucí velikostí objektu bude vzhledem k menšímu počtu komunikací a volání callbacků migrace dat rychlejší. Rychlost migrace by tedy měla být přímo úměrná velikosti objektu. Cílem experimentu je tyto hypotézy ověřit.

6.1.1 Metoda

Abychom byli schopni efektivně měřit přesun objektů, musíme v modelu vyvolat situace, kdy dochází k masivním přesunům. Jedním z nich je běh se sekvenčním I/O, tedy situace, kdy master proces sbírá ode všech ostatních data po jednotlivých objektech, aby je mohl zapsat do souboru. Byla proto měřena právě tato situace. Aby nebyl výpočet ovlivněn samotným I/O, skutečný zápis do souboru po sesbírání dat byl vynechán, takže naměřené hodnoty jsou ovlivněny pouze propustností MPI komunikací.

S tímto nastavením byla provedena sada běhů se stejnou konfigurací na obou clusterech. Ta byla následující:

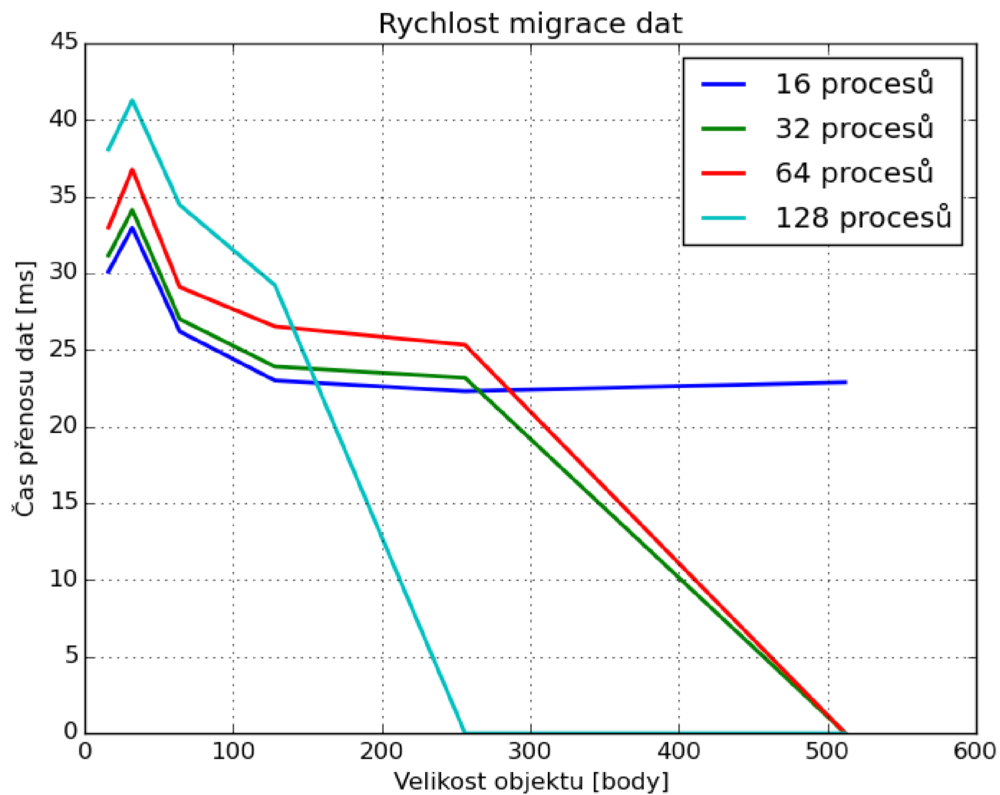
- Velikost domény: 2048
- Počet procesů: 16-128
- Velikost objektů: 16 - 512
- Počet iterací: 10 000

- Hustota zápisu: 10, tj. celkem 1000 zápisů

Horní hranice velikosti objektu byla dána počtem procesů, tedy velikostí bloků které vzniknou proto, aby velikost objektu nepřesáhla velikost bloku. Spodní hranice byla zvolena fixně s ohledem na velikost domény. Velikosti objektů, které pro daný počet procesů překročily maximum mají v grafu nulovou hodnotu.

6.1.2 Anselm

Na Anselmu byl naměřen průběh zobrazený na obrázku 6.1.



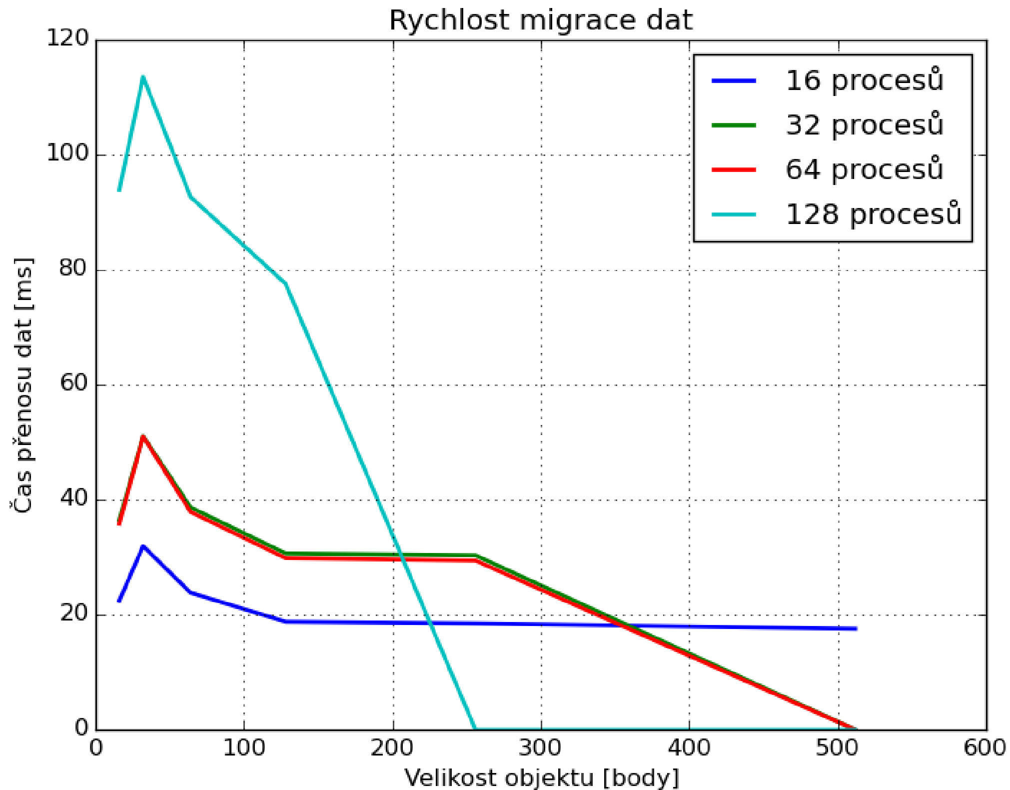
Obrázek 6.1: Graf rychlosti migrace dat - Anselm

Osa x představuje velikost objektu (stranu čtverce) v bodech domény a osa y představuje průměrný čas strávený přenosem celkem z 1000 iterací. Rozdílem oproti případu vyvažování je, že jde o sběr dat do master procesu, který může fungovat jako úzké hrdlo. V případě vyvažování se data vyměňují podle dekompozice mezi všemi procesy, potažmo uzly.

Z naměřených hodnot je patrné potvrzení hypotézy. S rostoucí velikostí objektu klesá doba nutná pro migraci dat. Výjimkou je rozdíl mezi velikostmi 16 a 32 bodů, kde je ve všech případech patrné drobné zpomalení. Přepočteme-li tuto situaci na velikost zpráv, dostáváme rozdíl 1024B a 4096B. Toto zpomalení je pravděpodobně způsobeno implementací Zoltanu. Ve všech ostatních případech je patrná klesající tendence, přibližně úměrná rozdílu velikosti objektů. Osa x udává pouze stranu čtverce, takže reálná velikost roste kvadraticky.

6.1.3 Salomon

Stejný experiment byl pro srovnání proveden na Salomonu, výsledek je v grafu 6.2.



Obrázek 6.2: Graf rychlosti migrace dat - Salomon

Oproti Anselmu je zde výrazný rozdíl v délce komunikací pro 128 procesů oproti zbytku. To může být způsobeno komunikacemi mezi více uzly, zvlášť vzhledem k tomu, že Salomon byl v době provádění experimentů poměrně vytížen. Vliv může mít také alokace vzdálenějších uzlů v topologii.

6.2 Dynamické vyvažování

Cílem těchto experimentů je ověřit efektivitu dynamického vyvažování, a to především zrychlení, jakého je schopen dosáhnout výpočet s vyvažováním oproti výpočtu bez vyvažování v případě zanesení umělého zpomalení. Případně jestli vůbec ke zrychlení dojde a v jakých situacích.

6.2.1 Metoda

Do výpočtu bylo zavedeno umělé zpomalení pomocí volání `std::this_thread:sleep_for`. Základ pro délku zpomalení je odvozen jako průměr běhu prvních pěti iterací. Předpokládáme, že je výpočet na začátku vyvážený, takže tato hodnota poslouží jako vhodný základ

pro zavedení konstantního zpomalení. Konečně hodnota slouží především pro srovnání vzhledem k délce běžné iterace.

Provedeny byly celkem 3 experimenty. Prvním z nich je ideální případ, kdy v 1/4 iterací dojde k zavedení zpomalení, které trvá až do konce simulace, 3/4 simulace jsou zpomalené. Detekce přitom proběhne v polovině výpočtu, takže druhá polovina je vyvážená. V tomto případě by se efektivita měla projevit nejvíce. Zároveň je tomto případu porovnáno škálování velikosti domény.

Druhý experiment již uvažuje případ, kdy zpomalení trvá kratší dobu. Zpomalení začne v 1/10 iterací a trvá po dobu 1/4 simulace, tedy do 35% iterací. Během výpočtu dojde ke třem detekcím v intervalu po 1/4 iterací. První detekce tedy systém vyváží a druhá a třetí jej vrací pomalu zpět do výchozího stavu.

Třetí experiment pracuje se zpomalením stejně jako druhý s tím rozdílem, že provádí celkem 10 detekcí v průběhu výpočtu. Při tomto počtu detekcí systém obvykle zkonverguje během simulace do výchozího stavu. Cílem je ověřit dopad vyšší režie na chování modelu.

Pro minimalizaci externích vlivů byly experimenty pouštěny bez výstupu do souboru.

Obecné parametry experimentů:

- Velikost domény: 8192^2
- Počet procesů: 16 - 128
- Velikost objektů: 32
- Počet iterací: 1 000
- Koeficienty zpomalení: 0,5; 0,75; 1; 1,5
- Práh detekce: 1,2

Koeficienty určují jaký podíl času iterace bude použit ke zpomalení. V případě 0,5, bude výpočet zpomalen o polovinu z vypočteného základu po dobu 25% nebo 75% běhu simulace. Celkovou dobu zpomalení tedy vypočteme jako:

$$delay = zaklad * koef * pocet_iteraci \tag{6.1}$$

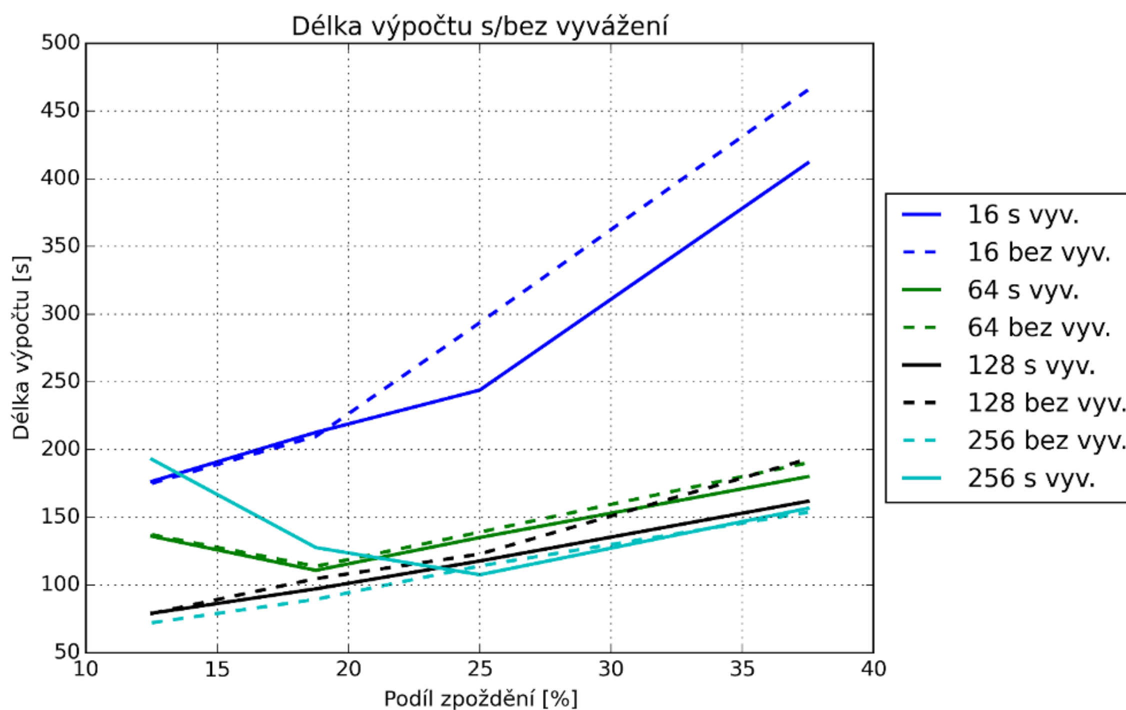
6.2.2 Ideální případ

Graf 6.3 představuje výsledky prvního experimentu s ideálním zpomalením. Osa x představuje podíl zpoždění získaný podle rovnice 6.1. Z grafu je patrné, že se hypotéza ve většině případů potvrdila. Vyvažované běhy (plná čára) jsou většinou rychlejší než běhy bez vyvážení. U běhů na 64 a 256 jádrech se při nižším podílu zpoždění projevuje režie, která převyšuje efektivitu.

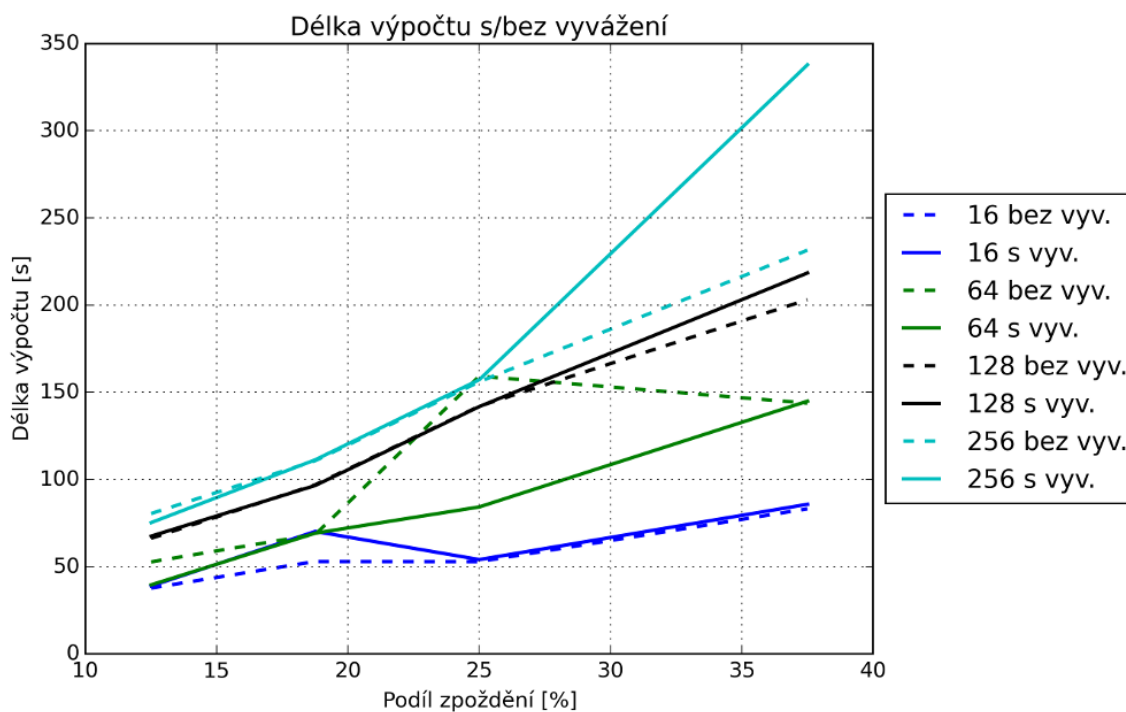
Vliv škálování

Efektivita vyvažování je závislá na velikosti domény. Pokud provádíme výpočet na příliš malé doméně, kde jsou jednotlivé iterace velmi krátké, převáží režie efekt vyvažování. To demonstruje následující experiment. Ten proběhl s podobnými parametry jako ten předchozí. Jedná se opět o ideální případ běhu, avšak na doméně velikosti 1024^2. Aby se výpočet pohyboval v řádově stejném časovém rozsahu, bylo provedeno 20 000 iterací. Délce iterace je úměrné i zpoždění, které je odvozeno opět stejným způsobem jako v předchozím experimentu. Pokud by bylo zpoždění neúměrně dlouhé k délce výpočtu, efekt vyvažování by

byl větší. S drobnou výjimkou v případě 64 procesů je z grafu 6.4 patrné, že čas výpočtu vyvažované verze je podobný, případně horší než bez vyvažování.

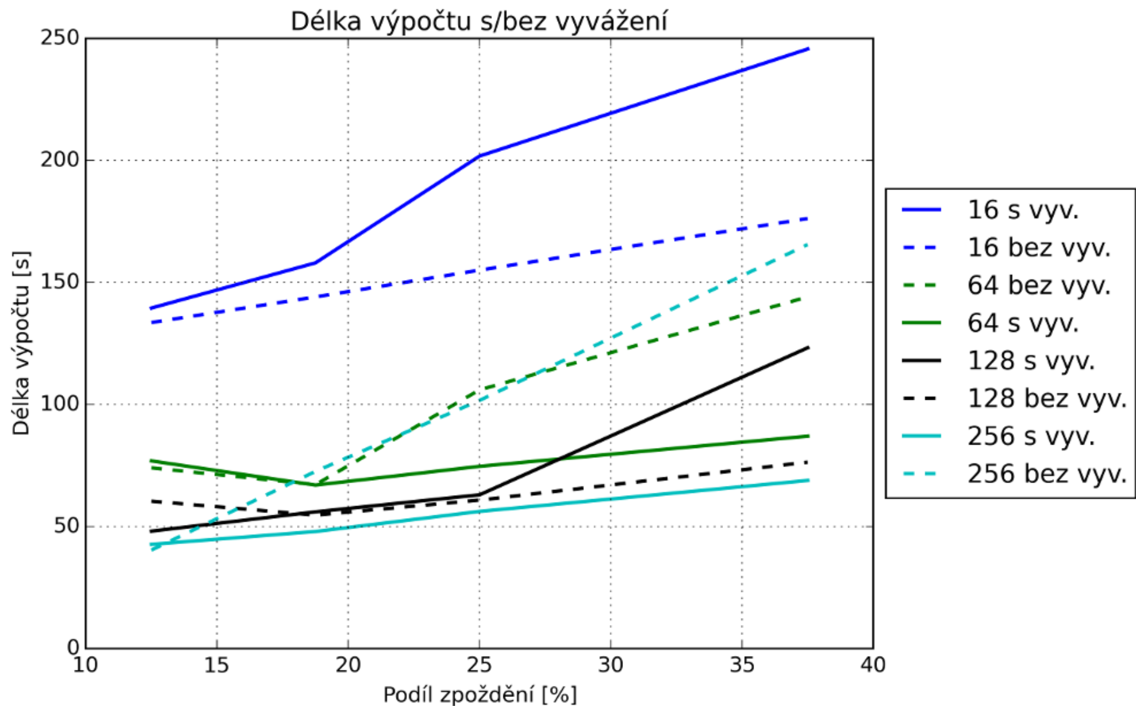


Obrázek 6.3: Zrychlení dynamickým vyvažováním, ideální případ



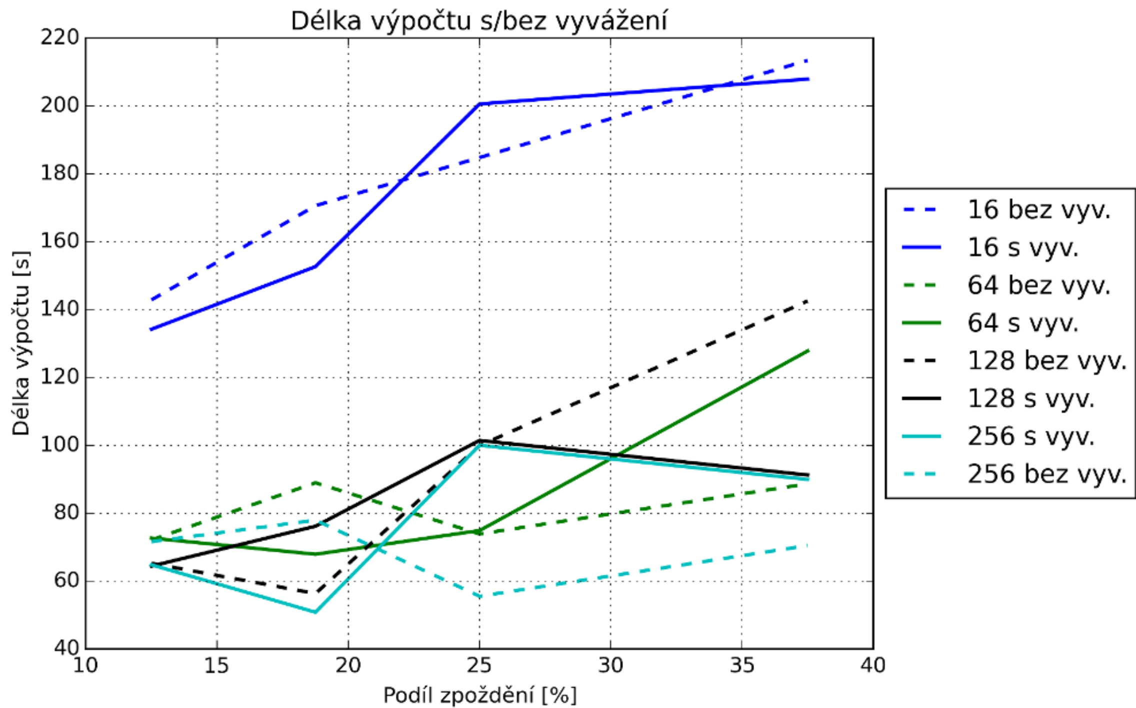
Obrázek 6.4: Zrychlení výpočtu dynamickým vyvažováním, doména 1024^2

Graf 6.5 představuje výsledky druhého experimentu. I v tomto případě je především pro 64 a 256 jader efekt vyvažování značný. U ostatních se však výrazně projevuje snaha o konvergenci do výchozího stavu, kdy dochází k přesouvání mnoha objektů mezi jednotlivými bloky a systém je tak po značnou část simulace nevyvážený. Samotné přemapování domény včetně migrace dat, zabral při těchto experimentech zhruba 1% - 3% času simulace.



Obrázek 6.5: Zrychlení výpočtu dynamickým vyvažováním, 3 detekce

Výsledky třetího experimentu jsou v grafu 6.6. Tady už je chování modelu komplikovanější. Projevují se zde některé nedostatky řešení. Systém je poměrně citlivý na přesnost měření. U těchto experimentů byl použit práh detekce nevyvážení 1,2, tedy pokud byl rozdíl nejkratšího a nejdelšího výpočtu větší než 1,2, byl výpočet vyvažován. U předchozích experimentů bylo žádoucí, aby systém vyvažoval při každé detekci. V tomto případě to už znamená vysokou režii a velkou dynamiku, která je z měření patrná.



Obrázek 6.6: Zrychlení výpočtu dynamickým vyvažováním, 10 detekcí

V grafu jsou výrazně pomalejší peaky okolo 25%,které naznačují hranici zpoždění, při které je konvergence do výchozího stavu nejnáročnější, tedy že se topologie během vyvažování velmi dynamicky mění.

Známé problémy

Experimenty nebyly prováděny na 32 procesech, protože model v tomto případě vykazuje nestabilní chování. Dalším významným problémem je, že model nezvládá pracovat s doménami nad 8192^2. Větší domény způsobují problémy s pamětí, které se bohužel nepodařilo odladit.

6.3 Zhodnocení výsledků

Celkem byly provedeny 4 experimenty. První z nich byl zaměřen na výkon systému v přesouvání dat pomocí Zoltanu. Měřena byla rychlost sběru dat ze všech ranků do ranku 0. Tento experiment potvrdil hypotézu, že s rostoucí velikostí objektu klesá doba potřebná k přesunu dat. Výjimkou byly velikosti 16 a 32 bodů, kde ve všech měřeních bylo 16 bodů rychlejší.

Další 3 experimenty již byly zaměřeny na dynamické vyvažování. V druhém experimentu šlo o ideální případ, který měl demonstrovat situace s minimální režií a maximálním vyvažovacím efektem. I zde se potvrdila efektivita vyvažování. Výjimkou byl běh s 256 procesy, kde je zjevné, že režie procesů převažuje efekt vyvažování. Pro 256 procesů by bylo třeba škálovat na větší doménu. Během tohoto experimentu se výpočet zrychlil v nejlepším případě o 16% při 25% zpoždění, což je poměrně dobrý výsledek.

Třetí experiment demonstroval stav, kdy byla do ideálního případu zanesena režie v podobě dalších dvou vyvážení. Zde dosáhly zajímavých výsledků především běhy s 64 a 256 procesy. Jednou z příčin mohou být značné rozdíly v dekompozici během jednotlivých vyvážení, kdy systém prochází několika nevyváženými stavy a tím je nevyvážená i velká část výpočtu.

Poslední experiment ověřoval schopnost systému konvergovat do výchozího stavu po ukončí zpomalování. Vyvažování bylo prováděno po 10% iterací. Každý z měřených zkonvergoval před ukončením simulace. Běhy s více procesy a tudíž menšími bloky konvergovaly obecně rychleji. Během experimentu se projevily značné fluktuace v dekompozici, což také zvýšilo efekt nevyvážených stavů.

Provedené experimenty demonstrovaly vyvažování na různých scénářích a v mnoha případech potvrdily jeho úspěšnost. Byly bohužel omezeny poměrně malou doménou. V zásadě však potvrdily výchozí hypotézy. Největším nedostatkem systému je poměrně vysoká citlivost na změny a z toho vyplývající nevyvážený výpočet během konvergence. Základní případy nevyvážení však systém řeší poměrně obstojně.

Závěr

Cílem této práce bylo navrhnout a implementovat systém pro dynamické vyvažování zátěže do paralelního modelu šíření tepla v chladiči procesoru. V rámci práce jsem se seznámil s architekturou superpočítačů Anselm a Salomon a dále jsem nastudoval potřebné techniky programování v paralelním prostředí s využitím knihovny MPI. Dále jsem prostudoval problematiku dynamického vyvažování zátěže a klíčových problémů, které s ní souvisí. Na to jsem navázal studiem současných přístupů k jejich řešení.

Význam použití těchto technik jsem demonstroval na statické variantě zmíněného modelu, která prokázala, že díky iterativiním datovým závislostem se zpoždění výpočtu jednoho z procesů, projeví odpovídajícím prodloužením celé simulace. Na základě této motivace jsem navrhl a implementoval dynamickou 2D dekompozici výpočetní domény a nad ní pracující algoritmus pro dynamické vyvažování.

Navržené řešení si kromě samotného vyvažování klade za cíl, posloužit jako experimentální platforma pro simulace podobného charakteru. Proto maximálně odděluje samotný výpočet od nutné režie. Lze jej poměrně snadno upravit pro použití jiných vyvažovacích algoritmů i jiných výpočetních metod. S tím souvisí i větší důraz na dokumentaci celého řešení.

Původním záměrem bylo využít ve vetší míře knihovnu Zoltan. Jednak pro práci s doménovými daty a jednak pro samotné vyvažování. Původně zvolený algoritmus RCB se bohužel ukázal jako nevhodný, protože nesplňoval omezení pro tvar subdomén. Proto jsem navrhl vlastní algoritmus, založený na geometrickém přístupu podobně jako zmíněný RCB. Algoritmus implementuje řádkové vyvažování ve 2D dekompozici, čímž kombinuje jednoduchost s využitím potenciálu použité dekompozice. Zároveň jsem navrhl systém detekce přetížení založený na měření času jednotlivých iterací simulace s možností využít statistické metody nad historií měření.

Experimenty ukázaly účinnost tohoto řešení v několika modelových situacích, kdy bylo dosaženo zrychlení okolo 16% oproti statické verzi. Zároveň byl demonstrován vliv škálování na účinnost vyvažování.

Možnosti pro další vývoj

Jedním z problémů řešení je citlivost na změny při práci nad již vyváženou doménou, a neefektivní konvergence do výchozího stavu. V této oblasti vzniká značný prostor pro další vývoj. Jednak v testování dalších scénářů a kombinací parametrů vyvažování nebo implementaci vhodnější řídící logiky.

Další experimenty mohou směřovat také k implementaci existujících algoritmů a jejich uzpůsobení dané doméně. To by mohlo vést k velmi zajímavým výsledkům.

Literatura

[1] Allinea Software: Allinea DDT: The debugger for C, C++ and Fortran threaded and parallel code. [online], březen 2017.
URL https://www.allinea.com/products/ddt

[2] Berger, M. J.; Bokhari, S. H.: A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, ročník 36, č. 5, Květen 1987: s. 570–580, ISSN 0018-9340, doi:10.1109/TC.1987.1676942.
URL http://dx.doi.org/10.1109/TC.1987.1676942

[3] Bretto, A.: *Hypergraphs: First Properties*. Heidelberg: Springer International Publishing, 2013, ISBN 978-3-319-00080-0, s. 23–42, doi:10.1007/978-3-319-00080-0_2.
URL http://dx.doi.org/10.1007/978-3-319-00080-0_2

[4] Catalyurek, U. V.; Boman, E. G.; Devine, K. D.; aj.: Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, ISSN 1530-2075, s. 1–11, doi:10.1109/IPDPS.2007.370258.

[5] Chevalier, C.; Pellegrini, F.: PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Comput.*, ročník 34, č. 6-8, Červenec 2008: s. 318–331, ISSN 0167-8191, doi:10.1016/j.parco.2007.12.001.
URL http://dx.doi.org/10.1016/j.parco.2007.12.001

[6] Devine, K.; Boman, E.; Heaphy, R.; aj.: Zoltan data management services for parallel dynamic applications. *Computing in Science Engineering*, ročník 4, č. 2, Mar 2002: s. 90–96, ISSN 1521-9615, doi:10.1109/5992.988653.

[7] Forum, M. P.: MPI: A Message-Passing Interface Standard. Technická zpráva, Knoxville, TN, USA, 1994.

[8] GitLab Inc.: Web projektu GitLab [online]. květen 2017.
URL https://about.gitlab.com/

[9] GWT-TUD GmbH: Vampir - Performance Optimization [online]. květen 2017.
URL https://www.vampir.eu/

[10] HDF5 Group: HDF5 Support page [online]. listopad 2016.
URL https://support.hdfgroup.org/HDF5/

[11] Heroux, M.; Bartlett, R.; Hoekstra, V. H. R.; aj.: An Overview of Trilinos. Technická Zpráva SAND2003-2927, Sandia National Laboratories, 2003.

[12] Lawrence Livermore National Laboratory: About VisIt [online]. listopad 2016.
URL https://wci.llnl.gov/simulation/computer-codes/visit

[13] Lengauer, T. (editor): *Combinatorial Algorithms for Integrated Circuit Layout*. Applicable Theory in Computer Science, Wiley, 1990.

[14] Meng, L.; Huang, C.; Zhao, C.; aj.: An improved Hilbert curve for parallel spatial data partitioning. *Geo-spatial Information Science*, ročník 10, č. 4, 2007: s. 282–286, doi:10.1007/s11806-007-0107-z, http://dx.doi.org/10.1007/s11806-007-0107-z.
URL http://dx.doi.org/10.1007/s11806-007-0107-z

[15] Nian, S.; Guangmin, L.: Dynamic Load Balancing Algorithm for MPI Parallel Computing. In *2009 International Conference on New Trends in Information and Service Science*, June 2009, s. 95–99, doi:10.1109/NISS.2009.171.

[16] Patzák, B.; Rypl, D.: Object-oriented, Parallel Finite Element Framework with Dynamic Load Balancing. *Adv. Eng. Softw.*, ročník 47, č. 1, Květen 2012: s. 35–50, ISSN 0965-9978, doi:10.1016/j.advengsoft.2011.12.008.
URL http://dx.doi.org/10.1016/j.advengsoft.2011.12.008

[17] Schloegel, K.; Karypis, G.; Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, ročník 14, č. 3, 2002: s. 219–240.
URL http://dblp.uni-trier.de/db/journals/concurrency/concurrency14.html#SchloegelKK02

[18] Trilinos Project: Trilinos/Zoltan2: Load Balancing and Combinatorial Scientific Computing:Doxygen documentation. [online], březen 2017.
URL http://www.cs.sandia.gov/~kddevin/papers/Dagstuhl09.pdf

[19] VI-HPS: Score-P:Scalable Performance Measurement Infrastructure for Parallel Codes. [online], květen 2017.
URL http://www.vi-hps.org/projects/score-p/

[20] Williams, R. D.: Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and experience*, ročník 3, č. 5, 1991: s. 457–481.

[21] Wu Yun-Long, Xu Xin-Hai, Yang Xue-Jun, Zou Shun, Ren Xiao-Guang: MDSLB:A new static load balancing method for parallel molecular dynamics simulations. *Chinese Physics B*, ročník 23, č. 2, 2014: 28903, doi:10.1088/1674-1056/23/2/028903.
URL http://cpb.iphy.ac.cn/EN/abstract/article_57907.shtml

Příloha A

Překlad Zoltanu

Knihovna Zoltan je distribuována v rámci balíku Trilinos. Zdrojové kódy lze získat z webu projektu [1].

Překlad projektu je řízen systémem cmake. Ten má řadu parametrů, vzhledem k velkému počtu modulů, které projekt obsahuje. Zoltan lze přeložit následujícím přikazem, který vytvoří standartní dynamicky linkovaný objekt libzoltan.so. Protože projekt nepodporuje překlad uvnitř výchozího adresáře, je třeba vytvořit složku např. build a v té příkaz spustit.

```
cmake -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF -D Trilinos_ENABLE_Zoltan:BOOL=ON -D
    TPL_ENABLE_MPI:BOOL=ON -D Trilinos_ENABLE_Fortran:BOOL=OFF -D
    CMAKE_INSTALL_PREFIX="some_path" -D BUILD_SHARED_LIBS=ON ../
```

Důležitým parametrem je *BUILD_SHARED_LIBS=ON*. V případě statického linkování knihovna vykazovala problémy se správou heapu v kombinaci s C++ kódem.

[1] https://trilinos.org/download/