VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

NEBLOKUJÍCÍ VSTUP/VÝSTUP PRO PROJEKT K-WAVE

NON-BLOCKING INPUT/OUTPUT FOR THE K-WAVE TOOLBOX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VÁCLAV KONDULA
AUTHOR

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

BRNO 2020

Ústav počítačových systémů (UPSY) Akademický rok 2019/2020

Zadání diplomové práce

21796

Student: **Kondula Václav, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Neblokující vstup/výstup pro projekt k-Wave**
Non-Blocking Input/Output for the k-Wave Toolbox
Kategorie: Paralelní a distribuované výpočty
Zadání:

1. Seznamte se se základními principy distribuovaného vstupu a výstupu v superpočítačových aplikacích. Prostudujte blokující i neblokující varianty. Zaměřte se především na knihovny MPI a HDF5.
2. Osvojte si základy práce se superpočítačovými systémy, prostudujte implementaci souborového systému Lustre a vstupně-výstupního subsystému projektu k-Wave.
3. Analyzujte výkonnost současné implementace vstupně-výstupního subsystému k-Wave v různých simulačních scénářích.
4. Na základně výsledků výkonnostní analýzy navrhněte neblokující vstupně-výstupní subsystém, který umožní překrytí výpočtu s ukládáním dat.
5. Navržená řešení implementujte.
6. Analyzujte dosažené výsledky a srovnejte je s původní implementací.
7. Zhodnoťte dosažené výsledky a diskutujte přínos navržené implementace pro praxi.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 25. října 2019

Abstrakt

Práce se zabývá implementací neblokujícího vstupně výstupního rozhraní pro projekt k-Wave, jež je navržen pro simulaci šíření ultrazvuku. Hlavní zaměření je na simulace velkých domén, jež kvůli vysokým nárokům na výpočetní výkon musí být spuštěny na superpočítačích a produkují až desítky GB dat během jediného simulačního kroku. V rámci této diplomové práce jsem navrhl a implementoval neblokující rozhraní pro ukládání dat využitím dedikovaných vláken, čímž se umožní překrytí výpočtu simulace s diskovými operacemi za účelem zkrácení doby provádění simulace. V projektu k-Wave se díky tomuto přístupu podařilo dosáhnout zrychlení až 33%, což má za následek mimo jiné také snížení finanční zátěže běhu simulace.

Abstract

This thesis deals with an implementation of non-blocking I/O interface for the k-Wave project, which is designed for time-domain simulation of ultrasound propagation. Main focus is on large domain simulations that, due to high computing power requirements, must run on supercomputers and produce tens of GB of data in a single simulation step. In this thesis, I have designed and implemented a non-blocking interface for storing data using dedicated threads, which allows to overlap simulation calculations with disk operations in order to speed up the simulation. An acceleration of up to 33% was achieved compared to the current implementation of project k-Wave, which resulted, among other things, also to reduce cost of the simulation.

Klíčová slova

k-Wave, MPI, I/O, C++, HDF5, PHDF5, paralelizmus, HPC, vícevláknové zpracování

Keywords

k-Wave, MPI, I/O, C++, HDF5, PHDF5, parallelism, HPC, multithreading

Citace

KONDULA, Václav. *Neblokující vstup/výstup pro projekt k-Wave.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jiří Jaroš, Ph.D.

Neblokující vstup/výstup pro projekt k-Wave

Prohlášení

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

. . . . . .
Václav Kondula
2. června 2020

Poděkování

Tato práce byla podpořena Ministerstvem školství, mládeže a tělovýchovy z Národního programu udržitelnosti II (NPU II) v rámci projektu IT4Innovations excellence in science - LQ1602 a výsledky byly získány s využitím výzkumné infrastruktury podpořené z programu Velkých infrastruktur pro výzkum, experimentální vývoj a inovace v rámci projektu IT4Innovations národní superpočítačové centrum - LM2015070. Také bych chtěl poděkovat vedoucímu práce doc. Ing. Jiřímu Jarošovi, Ph.D. za odbornou pomoc, cenné rady a věnovaný čas, jenž mi poskytl při vypracovávání této práce.

Obsah

1 Úvod **2**

2 Superpočítače **3**
2.1 Správce modulů EasyBuild . . . 3
2.2 Vstupně výstupní systém . . . 4
2.2.1 Souborový systém Lustre . . . 5
2.2.2 Souborový formát Hierarchical Data Format . . . 6
2.3 Profilovací a vývojové nástroje . . . 10
2.3.1 Arm DDT . . . 10
2.3.2 Arm MAP . . . 10
2.3.3 Score-P . . . 11
2.3.4 PAPI . . . 11

3 Paralelní zpracování **13**
3.1 Message Passing Interface . . . 14
3.2 Vícevláknové aplikace . . . 16
3.3 Vstupně výstupní jednotka procesoru . . . 17

4 Prototypová aplikace **18**
4.1 Analýza požadavků na zápis v projektu k-Wave . . . 18
4.2 Implementace neblokujícího zápisu . . . 20
4.3 Architektura . . . 21
4.4 Správa vláken . . . 23
4.5 Postup simulace . . . 26
4.6 Testování . . . 27

5 k-Wave **32**
5.1 Architektura . . . 32
5.2 Implementace neblokujícího zápisu . . . 35
5.3 Testování . . . 37

6 Závěr **45**

Literatura **47**

A Obsah DVD **50**

Kapitola 1

Úvod

Nároky na výpočetní výkon a objemy dat pro řešení problémů ve vědecké, inženýrské i podnikatelské sféře se stále zvyšují, avšak propustnosti perzistentních úložišť se nezrychlují dostatečným tempem. Běžně dostupné NVMe disky dosahují rychlosti zápisů až jednotek GB/s, avšak oproti pevným diskům nedosahují dostatečné spolehlivosti, jsou výrazně dražší a i přesto jsou řádově pomalejší oproti rychlostem procesorů. Vstupně výstupní systém se stává brzdícím prvkem způsobující výrazné zpomalení celé aplikace. Výrobci procesorů si uvědomují tento problém a řeší jej oddělením výpočetní a vstupně-výstupní jednotky. Vhodně navržené programy dokážou paralelně využívat obě jednotky a tím dosáhnout maximálního výkonu. Volně dostupné knihovny pro paralelní programování však nejsou na takové úrovni jako pro programování sériové a tedy je běžné využití blokujících vstupně výstupních operací. Tedy provádění programu je pozastaveno, dokud nebude vstupně výstupní operace dokončena, čímž je výrazně zpomaleno vykonávání programu.

Jedním z projektů, jenž vyžaduje práci s velkým objemem dat, je aplikace k-Wave. Slouží k provádění simulací šíření ultrazvuku ve 3-dimenzionálním prostoru a nalezla uplatnění především v medicíně pro simulaci ničení nádorů pomocí ultrazvuku. Pro malé domény je možné provádět simulaci na běžných počítačích s využitím verze aplikace k-Wave pro programovací jazyk Matlab. Pro simulace ve velkých doménách u reálných medicínských problémů je nutné využít paralelní MPI verzi navrženou pro spuštění na superpočítačích.

Jediná simulace dokáže vyprodukovat až jednotky TB dat (desítky GB dat v každém simulačním kroku) a při spuštění na 256 jádrech dosáhnout výpočetního času až 17 hodin. V současné verzi aplikace k-Wave probíhá zápis na konci každého simulačního kroku blokujícím způsobem a tedy zabírá značnou část běhu aplikace, čímž se prodlužuje délka běhu a v konečném důsledku také cena výpočtu.

V této diplomové práci jsem nejprve analyzoval prostředí superpočítačů, které je detailněji popsáno v kapitole 2, se zaměřením na vstupně výstupní systém. Následně v kapitole 3 jsem rozebral možnosti paralelizace k dosažení maximálního výkonu. Kapitola 4 se věnuje analýze požadavků na vstupně výstupní systém aplikace k-Wave. Na základě zjištěných poznatků jsem navrhl neblokující vstupně výstupní princip pro paralelní aplikace, jež využívá protokolu MPI a dedikovaných vláken pro vstupně výstupní operace. V první fázi vývoje jsem implementoval a otestoval prototypovou aplikaci využívající tohoto principu. S ohledem na výsledky testování jsem posléze implementoval tento princip také do samotné aplikace k-Wave verze MPI, čemuž se věnuje kapitola 5.

Kapitola 2

Superpočítače

Superpočítač je označení pro velmi výkonný počítač nejčastěji sestavený v podobě takzvaného clusteru. Cluster je seskupení jednotlivých uzlů, přičemž každý uzel připomíná architekturou běžný velmi výkonný počítač. Jednotlivé uzly jsou propojeny prostřednictvím vysokorychlostní počítačové sítě. Sestavený cluster tímto způsobem vyjde o mnoho levněji než jeden stejně výkonný počítač, zajišťuje vyšší dostupnost a umožňuje jednodušší alokaci a účtování zdrojů při práci mnoha uživatelů současně.

Tato diplomová práce byla vyvíjena a testována na superpočítači **Salomon** v Ostravě, který je spravován národním superpočítačovým centrem IT4Innovations. V době spuštění se jednalo o 40. nejvýkonnější superpočítač na světě dosahující výkonu až 2 PFLOPS[1]. Salomon se skládá z 1008 uzlů, z nichž 432 má navíc akcelerátor Intel Xeon Phi. Každý z uzlů má dva dvanácti jádrové procesory Intel Xeon a 128GB operační paměti. [18] Samotné uzly nemají vlastní perzistentní paměťové úložiště a jsou připojeny k síťovému souborovému systému Lustre, kterému je věnována kapitola 2.2.1. Lustre na Salomonu má kapacitu 1.6 PB (s limitem 100 TB na uživatele) a propustnost vstupně výstupních operací až 30 GB/s. [20]

Salomon v současné době slouží pro akademické i průmyslové účely a je tedy nutné spravedlivé rozdělení zdrojů mezi stovky uživatelů. Uživatele vkládají požadavky na provedení úlohy do zvolené front a úloha se provede asynchronně v okamžiku, kdy jsou dostupné požadované zdroje. Při vkládání úlohy do fronty si uživatel může zvolit mnoho parametrů jako je například počet jader, velikost operační paměti, přítomnost akcelerátorů a požadovanou délku běhu. Podle množství zdrojů si následně vybere vhodnou frontu. Expresní fronta povoluje maximální dobu běhu 1 hodinu a maximálně 8 uzlů. Pro náročnější úlohy je nutné zvolit produkční frontu. [19]

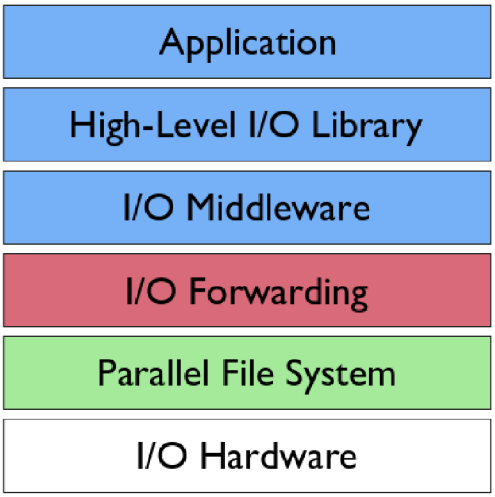
2.1 Správce modulů EasyBuild

EasyBuild je nástroj pro sestavení, instalaci a správu vědeckého softwaru navržený speciálně pro superpočítačové prostředí. [14] Instalace knihoven a nástrojů pomocí EasyBuild je automatizována, nevyžaduje superuživatelské oprávnění a dovoluje sdílení sestavených modulů. Superpočítače spravované IT4Innovations nabízí stovky již předinstalovaných modulů, jež jsou nejčastěji vyžadovány superpočítačovými aplikacemi. [17] K načítání a přepínání mezi moduly se využívá softwaru Lmod, jenž umožňuje instalaci více verzí stejného softwaru a jednoduché přepínání mezi nimi. Seznam modulů potřebných pro překlad a běh aplikace, jež je předmětem této diplomové práce, je přiložen v souboru Readme.

[1] Výkon 2 PFLOPS představuje $2 * 10^{15}$ operací čísel s plovoucí řádovou čárkou za sekundu.

2.2 Vstupně výstupní systém

Vstupně výstupní systém v superpočítačových prostředích má složitější strukturu než v tradičních počítačích. Zpravidla uzly nemají vlastní perzistentní paměťové úložiště a místo toho se využívá sdíleného síťového souborového systému. Vstupně výstupní systém musí tedy zajistit paralelní přístup všech uzlů do stejných paměťových míst. V superpočítačových prostředí se většinou využívá následují architektury uvedené na obrázku 2.1. [23]



Obrázek 2.1: Vrstvy vstupně výstupního systému v typických superpočítačových aplikacích [23]

- **Aplikace** je zpravidla navržena tak, jako by běžela na tradičním souborovém systému. Další vrstvy vytváří rozhraní, které je kompatibilní s POSIX standardem pro vstupně výstupní operace.
- **Vysokoúrovňová vstupně výstupní knihovna** poskytuje abstrakci nad prací se soubory a zajišťuje přenositelnost dat. Příkladem takové knihovny je HDF5, jíž je věnována sekce 2.2.2.
- **Vstupně výstupní mezivrstva** slouží k přeorganizování dat mezi procesy před zahájením vstupně výstupní operace. Například před zápisem dat z několika procesů do jednoho bloku paměti je vhodné předat všechna data jednomu procesu, které provede zápis, než aby se procesy zapisovali do bloku postupně. Příkladem takové knihovny je MPI, jíž je věnována kapitola 3.1.
- **Vstupně výstupní přeposílání** slouží k zachování koherence a konzistence dat v distribuovaných systémech. Tato vrstva odstiňuje souborový systém od paralelního výpočetního systému, přeskládává a agreguje dotazy a tím snižuje zátěž na souborový systém. Nejpoužívanějším otevřeným softwarem pro vstupně výstupní přeposílání je v současnosti IOFSL. [1]
- **Paralelní souborový systém** spravuje logický adresový prostor pro uložení dat a poskytuje přístup k datům. Nejvíce používanému souborovému systému Lustre je věnována následující sekce 2.2.1.

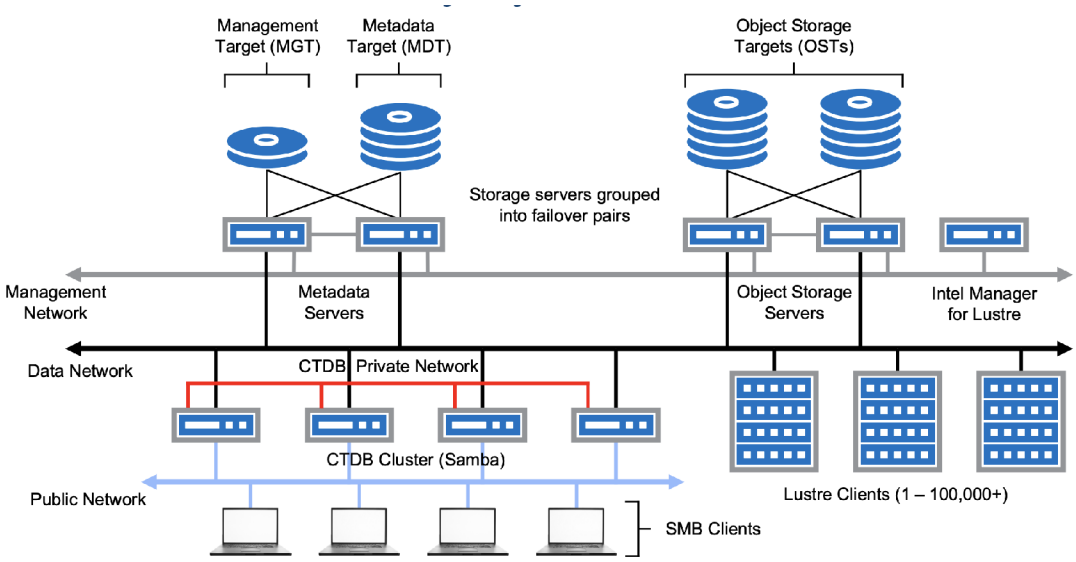
- **Hardware** pro perzistentní ukládání dat v superpočítačových systémech představují pole pevných a SSD disků. Pro archivaci a zálohování uživatelských dat se také používají magnetické pásky, ty však mají velmi dlouho přístupovou dobu a tedy nejsou vhodné pro přímý přístup.

2.2.1 Souborový systém Lustre

Lustre je distribuovaný, paralelní souborový systém navržený na masivní škálovatelnost, vysoký výkon a vysokou dostupnost. V současnosti Lustre využívá nadpoloviční většina z 500 nejvýkonnějších superpočítačů na světě. [27] Díky škálovatelné architektuře dovoluje práci současně až desetitisícům uživatelů a na nejrychlejších superpočítačích dosahuje průchodnosti vstupně výstupních operací až 1 TB/s. [28] Lustre je vydáván jako otevřený software pod licencí GNU General Public License.

Rozhraní souborového systému Lustre je kompatibilní s POSIXový standardem pro práci se soubory a adresáři. Z pohledu aplikace není nutné provádět žádné modifikace oproti práci se standardním linuxovým souborovým systémem jako je například ext4 nebo xfs.

Oproti standardním souborovým systémům, Lustre ukládá data a metadata na oddělené servery pro lepší škálovatelnost a dostupnost. Servery s metadaty udržují transakční záznamy změn metadat a podporují převzetí služeb při selhání, díky tomu výpadek sítě, jenž postihne jeden server s metadaty, neovlivní fungování celého souborového systému. [7]



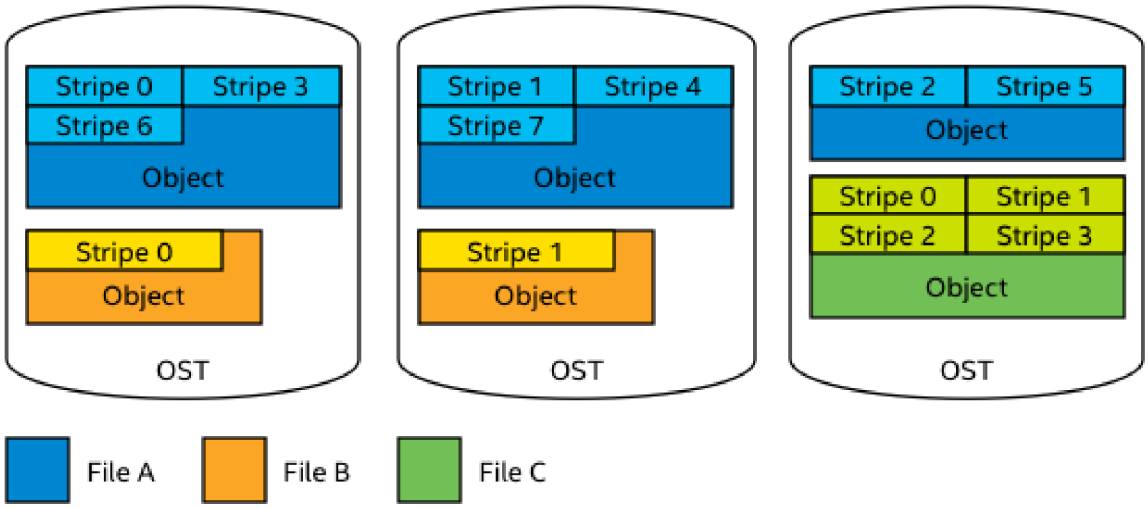
Obrázek 2.2: Architektura Lustre souborového systému [24]

Architektura Lustre vizualizována na obrázku 2.2 se skládá z následujících částí: [24]

- **Management service (MGS)** je globální řídící server, který může být společný pro více Lustre souborových systému, sloužící jako globální registr k uložení konfigurace Lustre systému, připojování klientů a registrování dílčích Lustre serverů. MGS se neúčastní vstupně výstupních operací.

- **Metadata service (MDS)** spravuje metadata uložená v paměti *Metadata Target (MDT)*. Mezi metadata patří například názvy souborů a adresářů, rozložení dat do oddílů (angl. Lustre stripes) a mapování oddílů na jednotlivé OSS. V rámci jednoho souborového systému je většinou zapojeno několik Metadata servis z důvodů rozložení zátěže, které jsou navíc párované k zajištění vysoké dostupnosti v případě výpadku.
- **Object storage service (OSS)** zpracovává požadavky na vstupně výstupní operace a poskytuje prostor pro uložení dat na *Object Storage Target (OST)*. Jeden soubor může být rozdělen mezi více OST (fyzických úložišť) po jednotlivých oddílech.
- **Client** je libovolné zařízení s nainstalovaným Lustre softwarem, jež pomocí protokolu Lustre Network protocol (LNet) propojuje Lustre souborový systém s operačním systémem. Klientské servery zpravidla nemají vlastní lokální perzistentní úložiště a připojují se k Lustre po síti.

Lustre klientský software obsahuje řadu nástrojů pro pro správu a monitorování souborového systému. Příkladem je `lfs` - klientský program na správu rozložení dat souborů. Pomocí příkazu `lfs getstripe -d PATH` je možné zobrazit výchozí nastavení velikostí Lustre stripů pro nové soubory v adresáři. Následně příkazem `lfs setstripe` můžeme změnit výchozí velikosti stripů tak, aby odpovídali násobkům velikosti bloků HDF5 souboru, čemuž je věnována další sekce. Obdobně je možné modifikovat velikosti stripů již pro existující soubor pomocí `lfs migrate`, avšak to vyžaduje přeorganizování souboru mezi OST a pro velké soubory je toto časově náročné.



Obrázek 2.3: Grafické znázornění rozložení souborů mezi OST [8]

Na obrázku 2.3 je naznačeno rozdělení tří souborů mezi tři OST. Soubory A a B jsou vhodně uloženy mezi více OST, avšak soubor C je celý uložený na jednom místě a tedy propustnost vstupně výstupních operací je limitovaná rychlostí jediného OST. Naopak čtení ze souboru A může teoreticky dosahovat až trojnásobné rychlosti.

2.2.2 Souborový formát Hierarchical Data Format

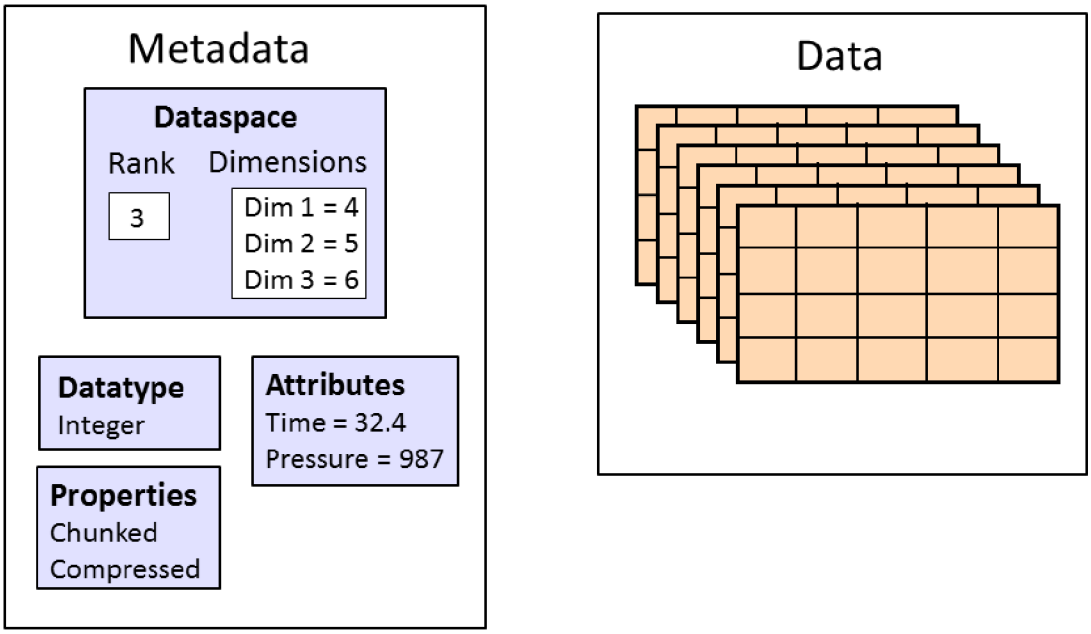
Hierarchický datový formát (angl. Hierarchical Data Format, dále jen HDF) byl vyvinut v americkém Národním centru pro superpočítačové aplikace (NCSA) a posléze převzat a

rozvíjen skupinou The HDF Group. Jedná se o nejrozšířenější standard pro ukládání velkých dat a tedy našel velké uplatnění i v superpočítačových aplikací. V současnosti nejnovější verze HDF5 se skládá ze tří částí:

- **souborový formát** pro ukládání dat
- **datový model** pro logickou organizaci dat a přístup k datům
- **software** zahrnující kolekci programů, knihoven, rozhraní a nástrojů pro práci s tímto formátem.

The HDF Group oficiálně podporuje pouze programovací jazyky C, C++ a Fortran. Avšak nad rámec softwaru vyvíjeného touto skupinou existuje velká řada komunitních projektů, jež mimo jiné obsahuje i knihovny pro práci s HDF pro většinu populárních programovacích jazyků. Mezi podporované jazyky díky komunitě otevřeného softwaru patří například Python, Matlab, Java, Go, R a Perl.

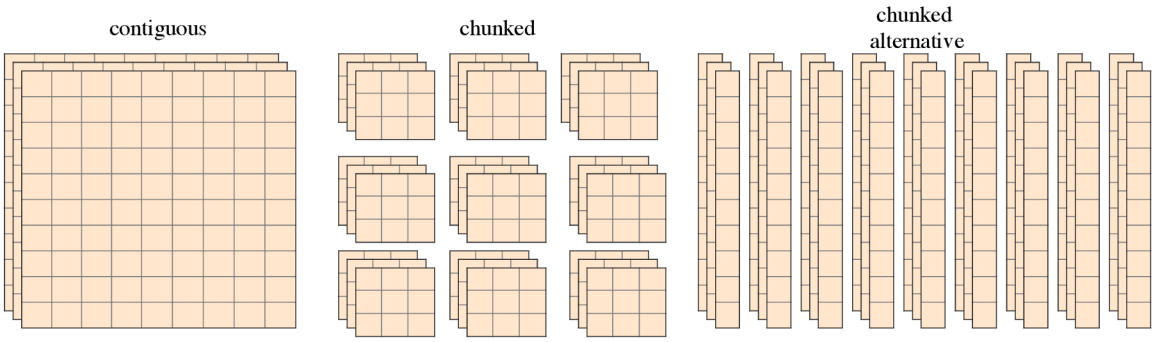
Datový model je ve verzi HDF5 zjednodušen oproti předcházejícím standardům, čímž se sjednotil přístup k různým typům objektů. Model se skládá pouze ze dvou primárních typů objektů. Prvním typem jsou **skupiny** (angl. groups), jež plní účel složek v tradičních souborových systémech. K uložení dat slouží **datasety**, jež se dále dělí na hlavičku a samotná data. Na obrázku 2.4 je zobrazen dataset obsahující trojrozměrné pole o pevné velikosti obsahující hodnoty datového typu *integer*.



Obrázek 2.4: Grafické znázornění hlavičky datasetu [33]

Hlavička datasetu se skládá ze čtyř části. **Dataspace** obsahuje informace o velikosti dat a počtu dimenzí. **Datový typ** určuje velikost a interpretaci uložených dat. HDF5 standard definuje vlastní datové typy, čímž je zajištěna kompatibilita a přenositelnost na rozdílných

architekturách. Kromě základních datových typů je možné vytvořit také vlastní složené datové typy a tedy ukládat data jako pole struktur. **Atributy** mohou byt přidruženy k datasetu a slouží k ukládání malých metadat. V příkladě na obrázku 2.4 jsou použity k zaznamenání hodnot o čase a tlaku pro dané měření. Na rozdíl od dat samotných neumožňují částečný zápis ani kompresi. **Vlastnosti** (angl. Properties) datasetu slouží k nastavení rozložení dat, tak aby byla vhodně uspořádána pro potřeby dané aplikace, a také komprese dat.



Obrázek 2.5: Příklad uspořádání dat pomocí chunků.

Rozložení dat v datasetu je ve výchozím nastavení ve formě souvislého pole. Tento způsob uložení dat je vhodný při přístupu do datasetu o statické velikosti po řádcích jedním procesem. Alternativním přístupem k rozložení dat je pomocí rozkouskování (angl. **chunking**). Dataset je rozdělen do stejně velkých polí (dále jen *chunks*) o stejném počtu dimenzí jako samotný dataset. Každý chunk je pak zvlášť komprimován a uložen do nezávislých paměťových prostor. Pořadí chunků v paměti nemusí reflektovat reálné pořadí v datasetu.

Rozdělení datasetu do chunků jde provést různými způsoby, jak je naznačeno na obrázku 2.5, avšak pokud není rozložení přizpůsobeno potřebám aplikace, může to mít negativní vliv na výkon. Při každém čtení i zápisu musí být vždy načten celý chunk. Pokud by jsme chtěli přečíst jeden řádek z posledního rozložení z obrázku 2.5, museli by jsme načíst do vyrovnávací paměti postupně celý dataset. Při paralelním programování je vhodné aby do jednoho chunku přistupoval zároveň nejvýše jeden proces, díky tomu nedochází k datovým konfliktům a procesy se navzájem neblokují.

V reálných aplikacích je vhodné vzít v úvahu nejen zápis, ale i následné čtení dat. Pokud generujeme data po řádcích, ale následně budou zpracovávány po sloupcích, pak buď můžeme zvolit velikost chunku jako kompromis pro oba přístupy, nebo optimalizovat velikost chunku pro zápis a následně pomocí nástroje *h5repack* [32] přeuspořádat rozložení dat.

Mezi nejčastější chyby při volbě velikosti chunků podle The HDF Group, jež mají negativní dopad na výkon aplikace, patří následující:

- **Velmi malé chunky:** Režie vyhledávání navazujících sekvencí dat je příliš vysoká vůči množství načtených dat a zároveň se zvyšuje počet vstupně výstupních operací, čímž se výrazně zpomaluje načítání dat.

- **Velmi velké chunky:** Pro přístup byť k jediné položce chunku je třeba načíst do vyrovnávací paměti celý chunk a dekomprimovat jej. Tento proces může být pro velká data pomalý a zároveň vyžaduje hodně prostředků. Pokud jsou chunky příliš velké může začít systém stránkovat na disk, čímž dojde ke zpomalení celého systému.

- **Malá vyrovnávací paměť:** Pokud iterujeme nad vyšším počtem chunků, než se vejde do vyrovnávací paměti, musí po každém kroku být některý chunk z vyrovnávací paměti zapsán na disk a nový nahrán do paměti. Zvyšuje se tím režie a v takovém případě je vhodnější vypnout vyrovnávací paměť pro dataset a zapisovat přímo na disk.
- **Nevhodná velikost hashovací tabulky:** Pokud se více chunků mapuje na stejné místo ve vyrovnávací paměti, pak nemohou být v paměti zároveň. Je proto vhodné zvolit dostatečnou velikost hashovací tabulky.

Rozhraní pro programování aplikací

HDF5 C knihovna se skládá z nízkoúrovňového a vysokoúrovňového rozhraní. [31] Nízkoúrovňové rozhraní umožňuje vysokou kontrolu nad HDF5 funkcionalitou a vysokoúrovňové rozhraní zjednodušuje práci s nejčastěji použitými funkcemi HDF5 knihovny. Zároveň tato knihovna obsahuje i rozhraní pro jazyk C++, jež je pouze zaobalením (angl. wrapper) funkcí v jazyce C.

Důležitým nízkoúrovňovým rozhraním jsou takzvané Property listy. [37] Poskytují mechanismus pro přidávání funkcionality pro volání HDF5 funkcí, aniž by zvyšovali počet argumentů dané funkce. Zároveň ulehčují práci programátorovi vyplněním nedůležitých (explicitně nenastavených) vlastností výchozími hodnotami.

Paralelní HDF5 (PHDF5) využívá k paralelnímu vstupně výstupním operacím knihovnu MPI, jíž je věnovaná kapitola 3.1. V následujícím krátkém návodu bude uveden popis vytvoření HDF5 souboru obsahující jeden dataset a následující zápis hodnot do tohoto datasetu. Názvy volaných funkcí a parametry odpovídají knihovním funkcím z jazyku C. Pokud není uvedeno jinak, při paralelní práci s HDF5 souborem se vždy účastní komunikace všechny procesy, které náleží do komunikátoru.

1. Nejprve je nutné vytvořit property list pro přístup k souboru pomocí `H5Pcreate(H5P_FILE_ACCESS)` a následně přiřadit do něj MPI komunikátor voláním `H5Pset_fapl_mpio`.
2. S tímto property listem následně vytvoříme prázdný soubor za použití `H5Fcreate`.
3. Inicializujeme property list pro dataset `H5Pcreate(H5P_DATASET_CREATE)` a pomocí `H5Pset_chunk` nastvíme, jak mají být data v datasetu uloženy do chunků.
4. Pro definování velikosti datasetu je třeba vytvořit dataspace voláním `H5Screate_simple`, čímž inicializujeme dataset na požadovanou velikost.
5. Vytvoříme samotný dataset voláním `H5Dcreate`.
6. Pokud vyžadujeme kolektivní zápis do datasetu, je nutné opět vytvořit property list pro zápis voláním `H5Pcreate(H5P_DATASET_XFER)` a zvolit kolektivní zápis voláním `H5Pset_dxpl_mpio(..., H5FD_MPIO_COLLECTIVE)`.
7. Každý proces si pomocí takzvaného *hyperslabu* zvolí část datasetu (podle velikosti bloku a offsetu), do kterého chce zapisovat, voláním `H5Sselect_hyperslab`.
8. Provedeme zápis voláním `H5Dwrite`.
9. Na závěr je nutné všechny property listy, dataspacy, datasety a soubory zavřít.

2.3 Profilovací a vývojové nástroje

Vývoj paralelních aplikací je nesnadný úkol. Proto si vývojáři pomáhají sadou nejrůznějších nástrojů, jež pomáhají nalézt chyby v programu, porozumět chování a docílit efektivnější implementace. Vzhledem k povaze aplikací, jež běží na superpočítačích, není většinou možné použít běžně rozšířené vývojové nástroje. Aplikace typicky pracují s velkým objemem dat, paralelně na více fyzických serverech a využívají velké množství procesů a vláken. Jednotlivé clustery v superpočítačových systémech také často využívají netypické hardware, jako je například akcelerátor Intel Xeon Phi dostupný na superpočítači Salomon.

V současné době již existuje řada nástrojů určena přímo na profilování a ladění náročných superpočítačových aplikací vyvíjených převážně univerzitami a výzkumnými centry, ale výjimkou nejsou ani nástroje vytvořené privátními organizacemi. Z nástrojů dostupných na superpočítači Salomon stojí za zmínku kolekce nástrojů vyvinuta společností Allinea Software, jež byla v roce 2016 přebrána skupinou Arm Holdings. [9] Nově je tato kolekce nástrojů vydávána pod obchodní značkou *ARM Forge*. Dalším užitečným nástrojem na profilování, trasování a analýzu kódu je nástroj *Score-P*, jenž je podporován řadou nadstavných analytických nástrojů s grafickým uživatelským prostředím.

2.3.1 Arm DDT

Arm Distributed Debugging Tool (DDT) je distribuovaný ladící nástroj, vydávaný součástí balíků Arm Forge. [3] Ladící nástroje slouží k vykonávání instrukcí programu místo vykonávání přímo na procesoru. Díky tomu je možné program kdykoliv pozastavit, krokovat jednotlivé příkazy, zobrazit hodnoty proměnných, popřípadě je přímo modifikovat. Nástroj Arm DDT využívá debuggovací informace dodávané překladačem a tedy je nutné při překladu použít přepínač `-g`. Podporovány jsou oba nejpoužívanější překladače: GNU GCC i Intel icc. Při překladu je také vhodné použít přepínač `-O0`, čímž překlač nepoužije žádné optimalizační kroky a tedy je zaručeno, že každý příkaz se vykoná v pořadí, v jakém je zadán ve zdrojovém souboru. V ukázce 2.1 je znázorněn postup příkazů pro překlad distribuované aplikace a spuštění Arm DDT na superpočítači Salomon.

```
1 $ module load Forge OpenMPI/4.0.0-GCC-6.3.0-2.27
2 $ mpicxx -g -O0 -std=c++11 main.cpp -o app
3 $ ddt mpirun ./app
```

Ukázka 2.1: Příkazy pro načtení modulů, překlad distribuované aplikace a spuštění ladícího nástroje Arm DDT.

2.3.2 Arm MAP

Arm MAP je distribuovaný profilovací nástroj, taktéž vydávaný součástí balíků Arm Forge. [4] Arm MAP byl první nástroj, jenž umožňuje profilování, analýzu a vizualizaci více procesových a vícevláknových aplikací. Díky možnosti profilování aplikací s výkonem až 1 ExaFLOP/s využívající až tisíce jader se stal velmi populární pro vývoj aplikací na největších superpočítačích.[5] Hlavními přednostmi Arm MAP je analýza zatížení jednotlivých jader, identifikace takzvaných spících vláken, jež zatěžují systém, podpora vektorizace na architekturách, jež vektorizací umožňují, a také analýza výkonu vstupně výstupních operací

na identifikaci úzkých míst. Profilování je možné v jak interaktivním módu, tak i v offline módu, jak je zobrazeno v ukázce 2.2

```
$ module load Forge OpenMPI/4.0.0-GCC-6.3.0-2.27
$ mpicxx -g -O3 -std=c++11 main.cpp -o app
$ # interactive mode
$ map mpirun ./app
$ # offline mode, followed by opening the results
$ map --profile mpirun ./app
$ map app_8p_1t_YYYY-MM-DD_HH:MM.map
```

Ukázka 2.2: Příkazy pro načtení modulů, překlad distribuované aplikace a spuštění profilovacího nástroje Arm MAP.

2.3.3 Score-P

Scalable Performance Measurement Infrastructure for Parallel Codes (Score-P) je vysoce škálovatelný nástroj pro profilování, trasování událostí a online analýzu superpočítačových aplikací vydaný pod BSD licencí. [13] Score-P využívá k uložení profilovacích informací otevřený formát Open Trace Format (OTF), díky čemuž je využíván jako instrumentační nástroj pro velkou řadu vysokoúrovňových analytických nástrojů jako jsou například Scalasca, Vampir, Periscope nebo Tau. Stejně jako nástroj Arm MAP umožňuje analýzu více procesových (MPI) a více vláknových (OpenMP) aplikací. Taktéž má podporu pro akcelerátory. Na rozdíl od Arm MAP, Score-P dodává instrumentační informace již při překladu aplikace a jsou obsaženy ve vygenerovaném binárním souboru, jak je znázorněno v ukázce 2.3.

```
$ module load Scalasca OpenMPI/4.0.0-GCC-6.3.0-2.27
$ scorep mpicxx -std=c++11 main.cpp -o app
$ scalasca -analyze -t mpirun ./app
```

Ukázka 2.3: Příkazy pro načtení modulů, překlad distribuované aplikace pomocí Score-P wrapperu a spuštění profilování pomocí nástroje Scalasca.

2.3.4 PAPI

Performance Application Programming Interface (PAPI) je nízkoúrovňové rozhraní ve formě knihovny, jež využívá zabudované hardwarové čítače moderních mikroprocesorů na měření výkonu. [30] Počet čítačů a i list dostupných událostí je závislý na architektuře procesoru, je tedy nutné přizpůsobit sledované události konkrétní architektuře. Seznam dostupných událostí je možné zobrazit pomocí aplikace `papi_avail`, popřípadě voláním nízkoúrovňového rozhraní přímo z analyzované aplikace. Často zkoumanými událostmi pomocí knihovny PAPI jsou například výpadky v jednotlivých úrovních paměti cache a počet operací s čísly s pohyblivou řádovou čárkou k získání informací o výkonu FLOP/s analyzované aplikace. Na rozdíl od nástrojů zmíněných v předchozích sekcích, PAPI vyžaduje změnu zdrojového kódu analyzované aplikace, čímž se jeho využití výrazně komplikuje. Zároveň však není vhodné kompilovat produkční verzi aplikace s knihovnou PAPI. Jednou z možností je využití preprocesoru překladače, jenž na základě zvolených parametrů vynechá knihovnu PAPI z překladu, jak je znázorněno v ukázce 2.4.

```
$ module load PAPI GCC
$ cat main.cpp

#include <iostream>
#ifdef WITH_PAPI
#include "papi.h"
#endif

int main() {
    #ifdef WITH_PAPI
        int num, retval;
        /* Initialize the PAPI library */
        retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT)
          exit(1);
        if ((num = PAPI_get_opt(PAPI_MAX_HWCTRS,NULL)) <= 0)
            exit(1);
        std::cout << "This machine has " << num << " hardware counters.\n";
    #endif
    std::cout << "Hello World!\n";
    return 0;
}
$ # compile and run without papi
$ g++ main.cpp -o app
$ ./app
Hello World!
$ # compile and run with papi
$ g++ -DWITH_PAPI=1 -lpapi main.cpp -o app
$ ./app
This machine has 11 hardware counters.
Hello World!
```

Ukázka 2.4: Překlad aplikace s využitím knihovny PAPI a bez ní.

Kapitola 3

Paralelní zpracování

Paralelizace, neboli souběžné provádění, se využívá pro zvýšení výpočetního výkonu, když sekvenční řešení již není dostačující. Kvůli fyzikálním limitacím současného designu tranzistorů využívaných ve výpočetních systémech, frekvence procesorů a tedy i rychlost sekvenčního zpracování se již nadále nezvyšuje takovým tempem, jakým doposud. [11] Výrobci hardwaru i programátoři softwaru proto v dnešní době pro zvýšení výkonu využívají paralelní zpracování.

Ne každý problém je však vhodné paralelizovat. V běžných aplikacích jsou data na sobě závislá a tedy není možné provádět libovolné operace paralelně. Paralelní program nikdy nemůže trvat kratší dobu, než nejdelší sekvence na sobě závislých operací. Pro výpočet předpokládaného zrychlení oproti sekvenčnímu řešení lze využít Amdahlova zákona (3.1).

$$F_P : \text{podíl paralelizovatelné části}$$

$$S_P : \text{zrychlení paralelizovatelné části}$$

$$S_{\text{celkové zrychlení}} = \frac{T_{\text{původní doba výpočtu}}}{T_{\text{zrychlená doba výpočtu}}} = \frac{1}{1 - F_P + \frac{F_P}{S_P}} \tag{3.1}$$

Ukázka 3.1: Amdahlův zákon [25]

Mějme například program, jenž 40% času provádí výpočet a zbytek času stráví režií a čekáním na vstupně výstupní operace, a spustíme jej paralelně na čtyřech jádrech. I při opomenutí přidané režie dosáhneme teoretického zrychlení maximálně o 43%, viz výpočet v ukázce 3.2.

$$S_{\text{celkové zrychlení}} = \frac{1}{1 - 0.4 + \frac{0.4}{4}} = 142.86\% \tag{3.2}$$

Ukázka 3.2: Amdahlův zákon aplikovaný na konkrétní případ

V paralelním provádění programu také přichází problémy, se kterými se nesetkáváme v sekvenčních programech. Pokud využíváme sdílené paměti (jak v operační paměti, tak na disku) může dojít k souběhu. Data jsou čtena a zapisována procesy v nesprávném pořadí což vede k nesprávným výsledkům programu. Řešením tohoto problému je využití kritické

sekce, čímž se zamezí souběžného přístupu k datům více procesy. To má však za následek zpomalení běhu.

Další negativním vlivem na výkon paralelních aplikací je synchronizace procesů. Procesy na sebe musí čekat, dochází k přepínání kontextu v procesoru a také část doby běhu je využit na komunikaci mezi procesy. Všechny tyto akce zvyšují režii. V určitých případech může dokonce být paralelní řešení pomalejší než sekvenční.

Paralelizace lze docílit na několika úrovních současně. V superpočítačových systémech, jimž je věnována kapitola 2, je v clusteru k dispozici až tisíce serverů přičemž každý server má několik jader. V takovém prostředí je vhodné využít paralelizaci na úrovní procesů pomocí protokolu MPI, které se věnuje sekce 3.1 na správu a synchronizaci procesů. V rámci jednotlivých procesů lze dílčí úlohy provádět více vlákny, jak je uvedeno v sekci 3.2. Na závěr v sekci 3.3 je popsán princip využití paralelizace na úrovní hardwaru procesoru souběžným využitím aritmetické a vstupně výstupní jednotky.

3.1 Message Passing Interface

Message Passing Interface (dále jen MPI) je otevřený síťový protokol pro paralelní řešení výpočetních problémů pomocí zasílání zpráv. Přestože MPI nebylo doposud standardizováno žádnou z předních standardizačně-vývojových organizací (jako je IEEE nebo ISO), stalo se nejrozšířenějším protokolem pro řešení paralelních problémů v superpočítačových systémech a je dostupné na všech předních superpočítačích. MPI definuje rozhraní i pro architektury se sdílenou pamětí, avšak mnohem častější uplatnění nalezlo v architekturách s pamětí distribuovanou.

Protokol MPI není závislý na architektuře ani na programovacím jazyku. V současnosti existují implementace například v jazycích C, C++, Java, Python, Fortran a pro zvýšení výkonu také přímo v hardwaru. Jednou z implementací protokolu je knihovna Open MPI, jež je kompatibilní s nejnovější verzí protokolu MPI3.1 a je součástí mnoha populárních distribucí Linuxu jako jsou Fedora, CentOS, Ubuntu a openSUSE. Knihovna Open MPI je také použita pro aplikaci, jež je předmětem této diplomové práce.

Komunikátor je základním objektem spojujícím skupiny procesů. Každé volání funkce z MPI knihovny pro zaslání zprávy vyžaduje jako parametr komunikátor, přes který bude zpráva poslána. Při spuštění aplikace jsou všechny procesy součástí komunikátoru `MPI_COMM_WORLD`. Procesy jsou v rámci komunikátoru identifikovány unikátním pořadovým číslem (angl. rank).

Pokud je komunikace vyžadována pouze pro podmnožiny procesů, musí se původní komunikátor rozdělit například voláním `MPI_Comm_split`. Každý z procesů se stane členem nového komunikátoru podle zadané barvy. Takto můžeme například vytvořit vlastní komunikátor pro každý výpočetní uzel. V nově vytvořeném komunikátoru mohou mít procesy jiné pořadové číslo než v původním komunikátoru podle zadaného parametru.

Point-to-point komunikace slouží pro zasílání zpráv mezi dvěma procesy. Typickým příkladem je zasílání dat, přičemž hlavní proces volá `MPI_Send` a druhý proces čeká na doručení dat voláním `MP_Recv`.

Kolektivní komunikace se účastní vždy všechny procesy v zadaném komunikátoru. Zpravidla všechny procesy volají stejnou funkci, avšak provedená akce záleží na pořadovém

čísle každého procesu. Například voláním `MPI_Bcast` hlavní proces (identifikovaný podle parametru `root`) rozešle stejná data všem ostatním procesům. Naopak voláním `MPI_Reduce` všechny procesy zašlou svoje data hlavnímu procesu.

Datové typy dat zasílaných ve zprávách pomocí MPI jsou standardizované z důvodu přenositelnosti a kompatibility různých architektur. Při zasílání dat se tedy vždy explicitně specifikuje datový typ definovaný knihovnou MPI. Díky tomu lze například spolehlivě komunikovat mezi dvěma uzly, přičemž jeden používá little-endian uspořádání bajtů a druhý big-endian.

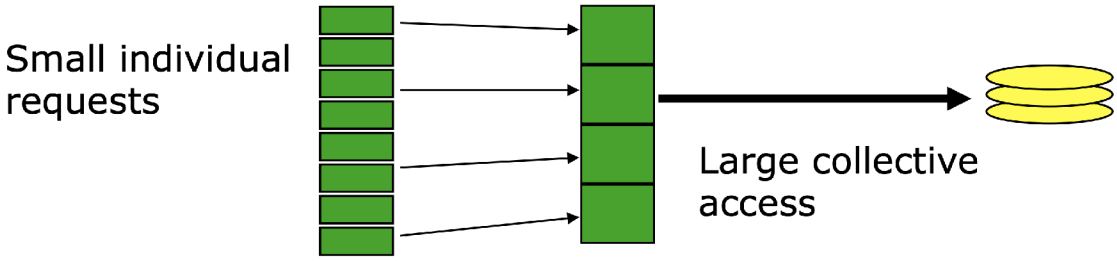
Blokující a neblokující zasílání

Mnoho funkcí pro zasílání zpráv jak pro point-to-point, tak pro kolektivní komunikaci má blokující i neblokující implementace. Například `MPI_Send` má pět základních módů pro zaslání zprávy: [2]

- **Standardní mód** (`MPI_Send`) nechává rozhodnutí, jestli bude zaslání bufferované, na knihovně MPI. Pokud vyžadujeme konkrétní implementaci je vhodné se tomuto módu vyhnout, jelikož se tento mód chová různě na rozdílných architekturách a při rozdílných velikostech odesílaných dat. Program může, ale nemusí pokračovat, nezávisle na tom, jestli přijímací proces obdržel data. Odeslání může proběhnout před tím, než přijímací proces je připraven data obdržet (zavolá *MPI_Recv*).
- **Bufferovaný mód** (`MPI_Bsend`) taktéž může proběhnout před tím, než přijímací proces je připraven data obdržet. Uživatel explicitně specifikuje buffer, který bude použit pro zaslání zprávy a musí zajistit, že nebude znovu použit nebo dealokován před skutečným odesláním zprávy. Pro kontrolu volnosti bufferu lze využít volání `MPI_Bsends`.
- **Synchronní mód** (`MPI_Ssend`) zajišťuje, že volání blokuje provádění procesu dokud přijímající strana skutečně neobdrží data. Díky tomu lze použít k synchronizaci procesů.
- **Ready mód** (`MPI_Rsend`) vyžaduje od uživatele, aby zajistil, že přijímající proces byl připraven přijmout data ihned v době volání. Tento mód je vhodný pokud vyžadujeme nejvyšší možný výkon, avšak ve většině případů je vhodnější použít synchronní mód.
- **Neblokující mód** (`MPI_Isend`) není nutně asynchronní. Volání funkce neblokuje a okamžitě se pokračuje ve vykonávání programu, uživatel je zodpovědný za to, že buffer nebude znovu použit nebo dealokován dokud data nebyla odeslána.

Vstupně výstupní operace

V paralelním programování na úrovni procesů jsou tři různé přístupy pro vstupně výstupní operace do souboru. [15] První je sekvenční, přičemž jeden proces sesbírá všechny požadavky na zápis od ostatních procesů a provede dotaz samostatně, čímž se znatelně zvyšuje režie operací. Druhým způsobem je zápis každého procesu do vlastního souboru, což zvyšuje režii na post-processing, pokud požaduje data v jednom souboru a tedy soubory se musí sloučit. V některých případech tento způsob není možný, pokud jsou data na sobě závislá.



Obrázek 3.3: Ukázka agregace kolektivních MPI dotazů na vstupně výstupní systém [15]

Součástí protokolu MPI je i standard pro vstupně výstupní systém (často označován jako *MPI I/O*), jenž přidává třetí způsob kolektivního zápisu do sdíleného souboru, čehož by nebylo možné standardním POSIX rozhraním bez MPI I/O dosáhnout. Jak je naznačeno na obrázku 3.3, MPI také agreguje malé požadavky na přístup do vstupně výstupního systému a tím výrazně snižuje režii operací.

Vstupně výstupní operace, obdobně jako operace pro zasílání zpráv, mají blokující i neblokující variantu. Pro práci s HDF5 soubory, jímž je věnována sekce 2.2.2, se využívá nadstavby HDF5 knihovny pro paralelní přístup PHDF5. [34]

3.2 Vícevláknové aplikace

Vlákna umožňují vykonávání konkurentních operací v rámci jednoho procesu. Oproti paralelizace na úrovní procesů je režie přepínání vláken daleko nižší, jelikož vlákna sdílí stejný adresový prostor a uživatelská oprávnění, tedy již není nutné je při přepíná měnit.

Implementace protokolu MPI verze 3.1 je již plně kompatibilní pro vícevláknové aplikace, které zároveň využívají více procesů. [36] Pro škálování rychlosti samotného výpočtu vlákna v multiprocesových aplikacích nepřináší zvýšení výkonu, jelikož zpravidla na každé jádro procesoru je již dedikován jeden proces, a tedy vlákna nemohou využít paralelizaci na úrovni hardwaru.

Vhodné použití vláken v multiprocesových aplikacích je na obsluhu periferií a síťových komunikací. Hlavní vlákno může být dedikované čistě na provádění výpočtu a nemusí být blokováno vstupně výstupními operacemi, které trvají řádově déle, než lokální operace. Vedlejší vlákna mohou sloužit k přednačítání a ukládání dat a také k synchronizaci mezi procesy.

Na podporu vícevláknových aplikacích se také zaměřili výrobci hardwaru. Nejznámější implementace z názvem *Hyper-Threading Techology (HTT)*, vyvinuta firmou Intel, [16] umožňuje souběžné vykovávaní více procesů na jednom fyzickém procesoru. Procesor podporující tuto technologii se jeví operačnímu systému jaké dva logické procesory, jež sdílí část fyzických prostředků. Souběžné provádění funguje pouze do okamžiku, než obě logická jádra chtějí použít stejný hardwarový prostředek. V ten moment se jeden z souběžných procesů pozastaví, dokud nebude prostředek volný. Druhé logické jádro tedy není plnohodnotným zdrojem a pokud operační systém nemá podporu pro HTT, je lepší tuto funkcionalitu vypnout, aby naopak nedošlo ke zpomalení systému. HTT je však ideální prostředek pro správu výpočetně nenáročných procesů a vláken, jež například obsluhují vstupně výstupní operace.

3.3 Vstupně výstupní jednotka procesoru

Rychlost zpracování strojových instrukcí procesoru je řádově vyšší než rychlost vstupně výstupních zařízení. Proto moderní procesory využívají k přenosu dat mezi operační pamětí a vstupně výstupním zařízením jednotku pro přímý přístup do paměti (angl. Direct Access Memory, dále jen DMA). [10]

Mnoho hardwarových systémů, jako jsou například řadiče disků, síťových karet a grafických karet, využívají DMA. Jelikož se jedná o oddělenou jednotku, procesor během čekání na vstupně výstupní operace může dále zpracovávat nezávislé instrukce a když jsou data připravena, DMA informuje procesor pomocí přerušení.

Problém nastává pokud knihovna využívající vstupně výstupní operace neumožňuje neblokující načítání a odesílání dat. Jedním z možných řešení je využití vláken, jak bylo naznačeno v předchozí sekci 3.2. Pro správu DMA se dedikuje vlákno, jež bude blokováno čekáním na vstupně výstupní operace, během čeho může hlavní vlákno provádět výpočet.

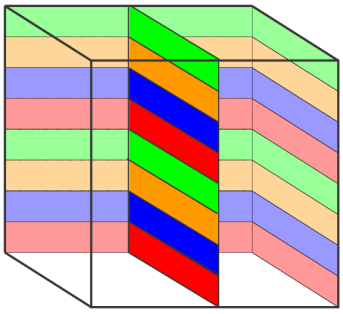
Kapitola 4

Prototypová aplikace

V první fázi vývoje jsem vytvořil prototypovou aplikaci, na níž jsem testoval překrytí neblokujícího zápisu do souboru s výpočtem. Díky tomuto izolovanému prostředí je možné simulovat chování neblokujícího zápisu při různém vytížení systému v jednotlivých simulačních krocích. Hlavními zkoumanými faktory je vliv vytížení operační paměti, vytížení sítě, rozložení dat mezi jednotlivé procesy, výpadky v jednotkách cache a navázání procesů a vláken na jednotlivá jádra.

4.1 Analýza požadavků na zápis v projektu k-Wave

Projekt k-Wave slouží k simulaci šíření ultrazvuku v 3-dimenzionálním prostoru. V MPI verzi projektu je doména – 3-rozměrné pole reprezentující zkoumaný prostor – rovnoměrně rozdělena mezi všechny procesy provádějící simulaci. V každém kroku simulace procesy vypočítají šíření ultrazvukových vln v postupně se zvyšujícím čase a výsledky se uloží kolektivním zápisem do souboru formátu HDF5, který je podrobně popsán v sekci 2.2.2. Není však nutné ukládat do souboru všechna data, ale pouze hodnoty ze zkoumaných subdomén. Podle rozložení ukládaných mezi procesy můžeme rozdělit zápis na dva typy:

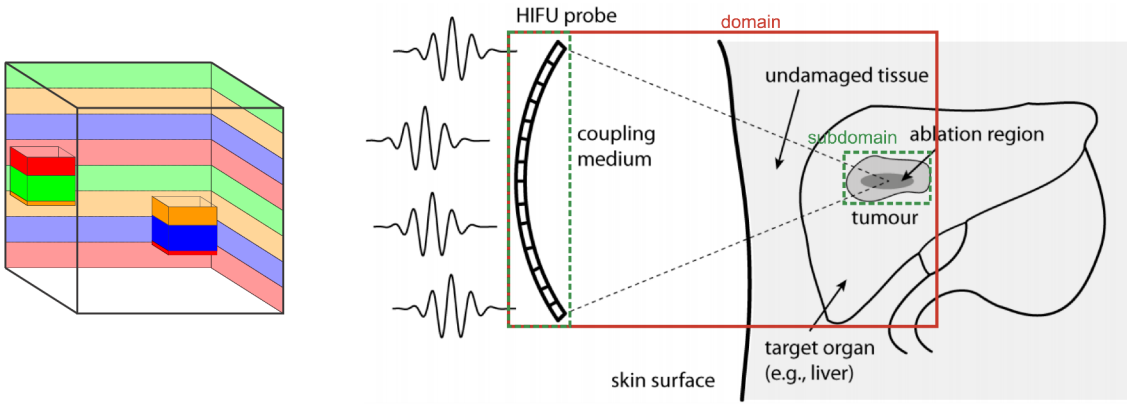


Obrázek 4.1: Segitální řez doménou, kde každý proces ukládá stejné množství dat

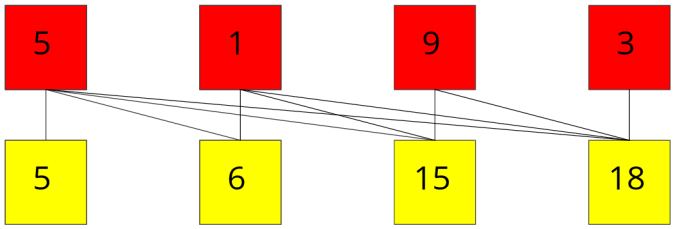
Řezy doménou rozdělují zapisována data rovnoměrně mezi všechny procesy. Každý proces si může vypočítat ofset lokace pro zápis svých dat podle svého pořadového čísla a tedy není nutné kolektivní MPI komunikací dopočítávat lokace pro zápis dat pro každý proces. Na obrázku 4.1 je znázorněn segitální řez doménou a rozdělení zapisovaných dat mezi

procesy. Jelikož známe předem velikosti dat, je vhodné uzpůsobit tomu velikosti chunků v datasetu a také velikosti lustre stripů pro soubor.

Kuboidy jsou subdomény ve tvaru kvádru sloužící k uložení dat ve zkoumaných oblastech. Na obrázku 4.2 je zobrazeno typické využití subdomén ve tvaru kuboidu k uložení dat v okolí vysílače a nádoru. Jak je také možno vidět z obrázku jednotlivé procesy (odděleny barevně) nemají shodné množství dat k uložení a některé procesy nemají data žádná. Před zahájením zápisu je tedy nutné vypočítat pro každý proces ofset, na kterém se budou data zapisovat, k čemuž je možné využít kolektivního volání funkce z knihovny MPI z názvem `MPI_Scan`. Jak je naznačeno na obrázku 4.3, každý proces odešle velikost dat, jež se chystá zapisovat, všem procesům s vyšším pořadovým číslem a následně si každý proces vypočítá ofset pro zápis svých dat.

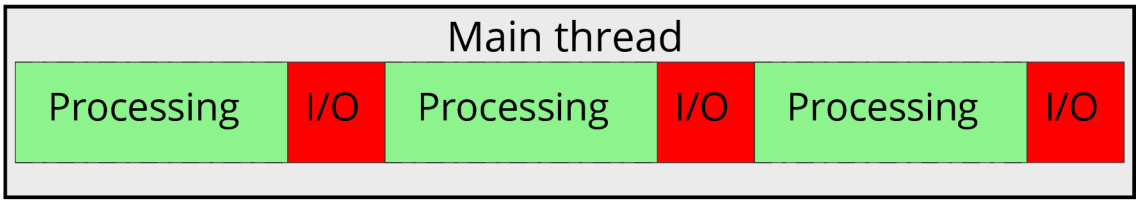


Obrázek 4.2: Výběr subdomén tvaru kuboidu podle umístění vysílače a nádoru [22]



Obrázek 4.3: Přeposlání velikosti dat, které každý proces musí uložit, pro výpočet ofsetu pomocí MPI_Scan

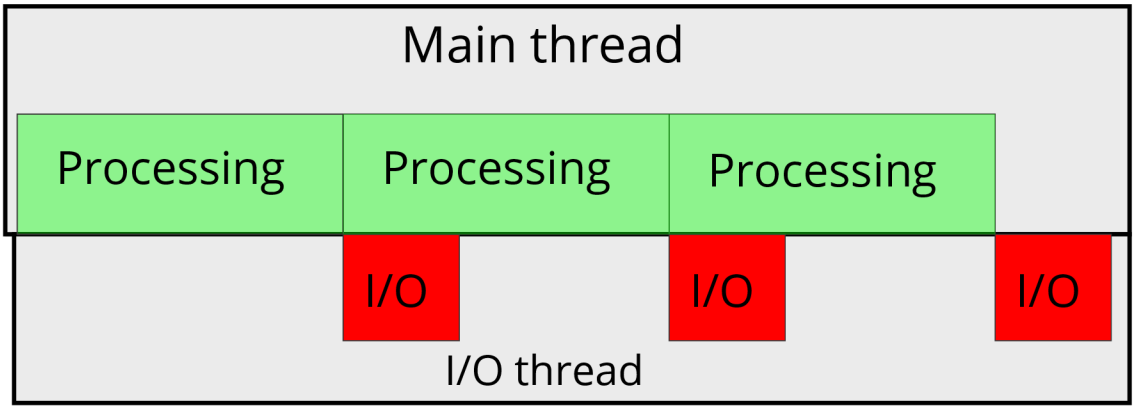
V současné MPI verzi aplikace k-Wave je zápis po každém kroku simulace prováděn blokující operací. Vstupně výstupní operace zabírají značnou část běhu aplikace, čímž se prodlužuje délka běhu a v konečném důsledku také cena. Během zápisů jsou jádra procesorů nevytížená, jelikož o správu vstupně výstupních operací se stará převážně jednotka DMA. Vizualizace kroků simulace a průběžného ukládání subdomén je znázorněna na obrázku 4.4.



Obrázek 4.4: Sériové zpracování v současné implementaci k-Wave

4.2 Implementace neblokujícího zápisu

Knihovna PHDF5 doposud neumožňuje neblokující zápis, a proto jsem v rámci této diplomové práce navrhl a implementoval rozhraní pro neblokující zápis pomocí vláken. Prototyp provádí simulační kroky obdobně jako reálná aplikace k-Wave, tedy v každém simulačním kroku se provádí časově náročný výpočet a po každé iteraci se ukládá obdobně velké množství dat (desítky GB). Zápis dat do souboru však není prováděn hlavními vlákny MPI procesů, ale pro každý MPI proces se vytvoří dedikované vlákno na vstupně výstupní operace. Tyto vlákna provádí všechny kolektivní zápisy do souboru, jak je znázorněno na obrázku 4.5.



Obrázek 4.5: Navržené paralelní zpracování implementované v prototypu

Limitace MPI rozhraní

Protokol MPI vyžaduje, aby se všechny procesy v rámci jednoho komunikátoru účastnili kolektivních blokujících operací v přesně daném pořadí. Pokud je tedy současně prováděn zápis z pomocných vláken a synchronizace hlavního běhu simulace, může dojít k uváznutí (angl. Deadlock). Proto rozhraní pro paralelní zápis využívá dedikovaný MPI komunikátor pro pomocná vlákna, aby nedocházelo ke konfliktům zasílaných zpráv.

Limitace PHDF5 knihovy

Současnou verze knihovny PHDF5 lze zkompilovat dvěma různými způsoby pro využití paralelního přístupu do souboru. [35] Pro účely projektu k-Wave je nutné využít při kompilování knihovny přepínače `-enable-parallel`, který umožní přístup do souboru z více MPI procesů pomocí kolektivních volání. S tímto nastavení není však možné provést více jak jeden přístup do souboru současně. I přesto, že jednotlivé přístupy do souboru se nijak nepřekrývají, například přistupují do různých datasetů nebo dokonce i do jiných souborů, knihovna HDF5 modifikuje globální proměnné a tudíž dochází k porušení ochrany paměti a následnému selhání aplikace.

Aby bylo možné volat z více vláken v rámci jednoho procesu, musí být knihovna zkompilována s přepínačem `-enable-threadsafe`. Tyto dva přepínače jsou však vzájemně výlučné, bez explicitního povolení nepodporovaných funkcí pomocí `-enable-unsupported`. Použití tohoto nastavení však značně omezuje přenositelnost programu. Zároveň povolení threadsafe přístupu nezajišťuje souběžný přístup do souboru. Je sice možné paralelně provést volání HDF5 knihovny, avšak tato volání jsou pouze serializovaná do fronty a jsou prováděna postupně. Rozhodl jsem se tedy využít pouze podporované funkcionality a zajistit bezpečný přístup k funkcím knihovny HDF5 ve vlastní režii.

4.3 Architektura

Architektura prototypové aplikace se skládá z několika částí, jež ve své podstatě kopírují strukturu MPI verze aplikace k-Wave. Samotný výpočet simulace je však výrazně zjednodušený a umožňuje jednoduché modifikace za účelem testování využití neblokujícího zápisu při různém zatížení systému. Zdrojové soubory prototypové aplikace jsou uloženy na paměťovém médiu A ve složce `/Sources`.

Hlavní běh simulace je implementován v souboru `main.cpp` a spojuje všechny následující komponenty dohromady. Postup simulace je popsán níže v sekci 4.5.

Vstupní doména je reprezentována třídou `SerialInputData` ze souboru `input_data.h`. Vstupní doména má tvar krychle o předem zadané velikosti a je rozdělena v ose Z rovnoměrně mezi všechny MPI procesy. Pro jednoduchou kontrolu korektního výpočtu, výběru a zápisu hodnot obsahují jednotlivé prvky trojrozměrného pole hodnoty určené funkcí (4.1). Na počátku každého simulačního kroku každý z procesů zvýší hodnotu prvků, jež danému procesu náleží, o třetí mocninu velikosti hrany krychle.

$$a : \text{délka strany krychle}$$

$$x, y, z : \text{souřadnice bodu v poli}$$

$$t : \text{krok simulace}$$

$$V_{x,y,z,t} = x + y * a + z * a^2 + t * a^3 \tag{4.1}$$

Ukázka 4.6: Určení hodnoty bodu `V` o souřadnicích `x, y, z` v simulačním kroku `t`.

Výběr subdomén je realizován v souboru `pattern.h`. Pro účely prototypové aplikace byly implementovány dvě třídy, jež obě dědí z virtuální třídy `BasePattern`. První třída `XCut` reprezentuje subdoménu řezu doménou v ose X. Každému z procesů tedy připadá stejné množství dat určených na zápis. Druhou třídou je `Cuboid`, jež reprezentuje subdoménu tvaru krychle. Jak bylo popsáno v sekci 4.1, procesy si před samotným uložení dat musí předat informaci, kolik každý z procesů bude ukládat dat, aby si procesy vhodně zvolily hyperslaby.

Každý proces v každém iteračním kroku prochází vstupní data, která mu náleží, a pro každý bod volá metodu `BasePattern::match`. Pokud bod náleží do subdomény (volání metody vrátí hodnotu `true`), bude na konci iteračního kroku bod uložen do výstupního souboru.

Parametry simulace je možné zadávat přes parametry příkazové řádky programu. K zpracování parametrů je v programu využita volně šiřitelná knihovna `CLI11` [29], jež je kompatibilní se všemi standardy jazyka C++ od verze C++11. Parametry jsou zpracovány všemi MPI procesy, avšak vypisování chybových hlášek a nápovědy je prováděno pouze jedním hlavním procesem, aby nedošlo k duplicitním výstupům. Pomocí parametrů simulace lze editovat například velikost vstupní domény, velikost a tvar zkoumané subdomény, počet iteračních kroků, velikost vektoru pomocných polí, velikost chunků v HDF5 souboru a zatížení systému v jednotlivých iteračních krocích, jak je znázorněno v ukázce 4.1.

```
1  $ mpirun ./aiompi --help
2  prototype application to test non-blocking write
3  Usage: ./aiompi [OPTIONS] output
4
5  Positionals:
6    output TEXT REQUIRED path to the output file
7
8  Options:
9    -h,--help Print this help message and exit
10   -s,--grid-size UINT:POSITIVE
11                              input domain edge size (default: 512)
12   -i,--iteration-count UINT:POSITIVE
13                              count of simulation iterations (default: 128)
14   -b,--buffer-count UINT:POSITIVE
15                              count of rotating buffers used for write (default: 3)
16   -c,--copy-count UINT:POSITIVE
17                              data multiplier (default: 1)
18   -w,--write-count UINT:POSITIVE
19                              write multiplier (default: 1)
```

Ukázka 4.1: Nápověda k prototypové aplikaci znázorňující dostupné parametry a jejich funkci.

Správa vláken je implementována v souboru `thread_pool.h`, jež byla částečně přejatá z volně šiřitelné knihovny *ThreadPool* [26] a obohacena o další metody nutné ke správnému běhu aplikace. Všechny mnou provedené změny jsou zaznačené ve zdrojovém souboru podle požadavků přiložené licence. Podrobnější popis správy vláken je uveden v následující sekci 4.4.

Správa pomocných polí je realizována jako šablona třídy `BufferManager` v souboru `buffer.h`. Parametrem šablony je datový typ, jemuž odpovídají jednotlivé položky v polích. Parametrem pro konstruktor je pak počet rotujících se polí. Pro účely prototypové aplikace je třeba tři instance této třídy. První instance s poli o datových typech `unsigned long` slouží

k uložení dat, jež mají být zapsána do souboru. Další dvě instance s poli o datových typech `hsize_t` slouží pro uložení rozměrů hyperslabu a jeho posunutí vůči začátku datasetu.

Na začátku každého iteračního kroku si hlavní vlákno vyžádá volné pole voláním `BufferManager::get_buffer_index`. Pokud žádné pole není k dispozici, hlavní vlákno čeká, než je pole uvolněno zapisujícím vláknem. Metoda vrátí index pole do vektoru polí, jež je rezervováno pro daný simulační krok. Hlavní vlákno následně do vektoru `BufferManager::futures` na pozici daného indexu vloží instanci `std::future` asynchronní operace, jež pole využívá. Pole je považováno za volné a připravené k znovu využití v momentě, kdy daná instance `std::future` vrátí hodnotu.

Zatížení systému slouží k testování překrytí asynchronního zápisu s běžnou zátěží v rámci jednotlivých iteračních kroků. K dispozici jsou tři různé testovací scénáře, jejích účel je navodit stav různého vytížení systému. Každý ze scénářů je konfigurovatelný pomocí parametrů simulace. K vytížení síťových služeb slouží metoda `broadcast`, jež využívá volání `MPI_Bcast` k přeposílání velkého množství dat z každého MPI procesu na všechny ostatní. Vytížení hlavní paměti je implementováno ve funkci `memoryHeavy`. Pomocí standardní funkce `memcpy` jsou kopírována data z jednoho pole do druhého. Efektivní vytížení procesoru je implementováno ve funkci `matrixMultiplication`, v níž je prováděno násobení matic. Pro testování je vhodné nastavit velikost matic tak, aby se vešly do paměti cache. V opačném případě způsobí výpadky paměti cache natažení dat z hlavní paměti a výsledky testování mohou být kvůli tomu zkreslené.

Profilování a měření výkonu - K měření jak celkové doby vykonávání programu, tak dílčích částí simulace jsem využil volání `MPI_Wtime`, jež vrací uplynulý čas na volajícím procesoru. K nízkoúrovňovému profilování aplikace jsem využil knihovnu PAPI popsanou v sekci 2.3.4. K monitorování PAPI událostí je nutné přeložit prototypovou aplikaci s nastavenou proměnnou prostředí `WITH_PAPI=1`, čímž preprocesor překladače zahrne také základní knihovnu PAPI a také rozšíření `PAPI wrapper` vyvinuté výzkumnou skupinou Swiss National Supercomputing Center (CSCS). Následně je možné specifikovat měřené PAPI události prostřednictvím proměnného prostředí `PAPI_EVENTS` při spuštění prototypové aplikace. Pro monitorování více událostí, oddělujeme názvy jednotlivých událostí znakem svislé čáry (|). Ukázka 4.2 obsahuje příkazy pro překlad a spuštění prototypové aplikace za účelem monitorování výpadků v L2 cache.

```
1 $ make clean
2 $ WITH_PAPI=1 make
3 $ PAPI_EVENTS="PAPI_L2_TCM|PAPI_L2_TCA" mpirun ./aiompi
```

Ukázka 4.2: Příkazy pro překlad a spuštění prototypové aplikace za účelem monitorování výpadků v L2 cache.

4.4 Správa vláken

Vytváření nových vláken je operace s velkou režií. Není tedy vhodné vytvářet nové vlákno na provedení zápisu pro každou ze stovek iteračních kroků. Rozhodl jsem se využít existující implementaci pro provedení asynchronních úkonů *ThreadPool* [26].

Každý z procesů si na začátku vykonávání programu vytvoří jedno vlákno, které bude obsluhovat frontu pro asynchronní zápis. Toto vlákno čeká na vložení požadavků na zápis do souboru hlavním vláknem. Jelikož je fronta obsluhována pouze jedním vláknem a zápisy jsou prováděny v pořadí, nemůže dojít k souběžným přístupům do souboru. Tím je zajištěn bezpečný přístup k prostředkům knihovny HDF5 a není nutná explicitní threadsafe podpora v knihovně HDF5.

Hlavní vlákno si na začátku každého iteračního kroku rezervuje pole pro data určena k zápisu a následně vloží požadavek do fronty pro zápis. Po provedení zápisu je pole opět uvolněno pro znovupoužití.

Režie vláken

Díky využití perzistentních pomocných vláken pro zápis, je režie vytvoření a ukončení vlákna minimální. Avšak přidaná režie se projeví i při jednotlivých iteračních krocích.

Čekání na požadavek ve frontě - Vlákna využívají k synchronizaci standardní posixový zámek (angl. mutex) a podmíněnou proměnou (angl. conditional variable). Když vlákno ukončí požadavek na zápis, zamkne zámek a zkontroluje frontu, jestli se v ní nachází další požadavky. Pokud ano, odebere jej z fronty, odemkne zámek a okamžitě jej provede. V opačném případě začne čekat na podmíněné proměnné, čímž se také odemkne zámek. V momentě když hlavní vlákno vloží nový požadavek na zápis do fronty, informuje pomocné vlákno čekající na událost na podmíněné proměnné, čímž pomocné vlákno probudí. Tento přístup synchronizace má minimální režii v porovnání s jinými synchronizačními postupy.

Výpadky v paměti cache - Jak je patrné z předchozí sekce 4.2, v současné době je velmi malá podpora knihoven pro využití kombinace více vláken a více procesů v rámci jednoho programu. Bohužel z toho důvodu dosud žádný ze standardních profilovacích nástrojů neumožňuje analýzu takových aplikací. Z toho důvodu jsem se rozhodl využít nízkoúrovňovou knihovnu PAPI, popsanou v sekci 2.3, pro měření výpadku cache. Jak je vidět v tabulce 4.1, výpadky v cache způsobené paralelizací zápisu mají minimální vliv na výkon programu.

Event	Blocking write	Non-blocking write
PAPI_L1_TCM	478 383 552	478 553 957
PAPI_LD_INS	14 327 092 213	14 327 513 327
PAPI_SR_INS	4 849 400 418	4 849 415 515
PAPI_L2_TCM	240 785 671	242 806 696
PAPI_L2_TCA	478 445 478	478 621 821
PAPI_L3_TCM	78 546 020	77 098 428
PAPI_L3_TCA	240 785 671	242 806 696
L1 cache miss	**2.49%**	**2.49%**
L2 cache miss	**50.3%**	**50.7%**
L3 cache miss	**32.6%**	**31.8%**
Memory Bandwidth	288MB/s	293MB/s

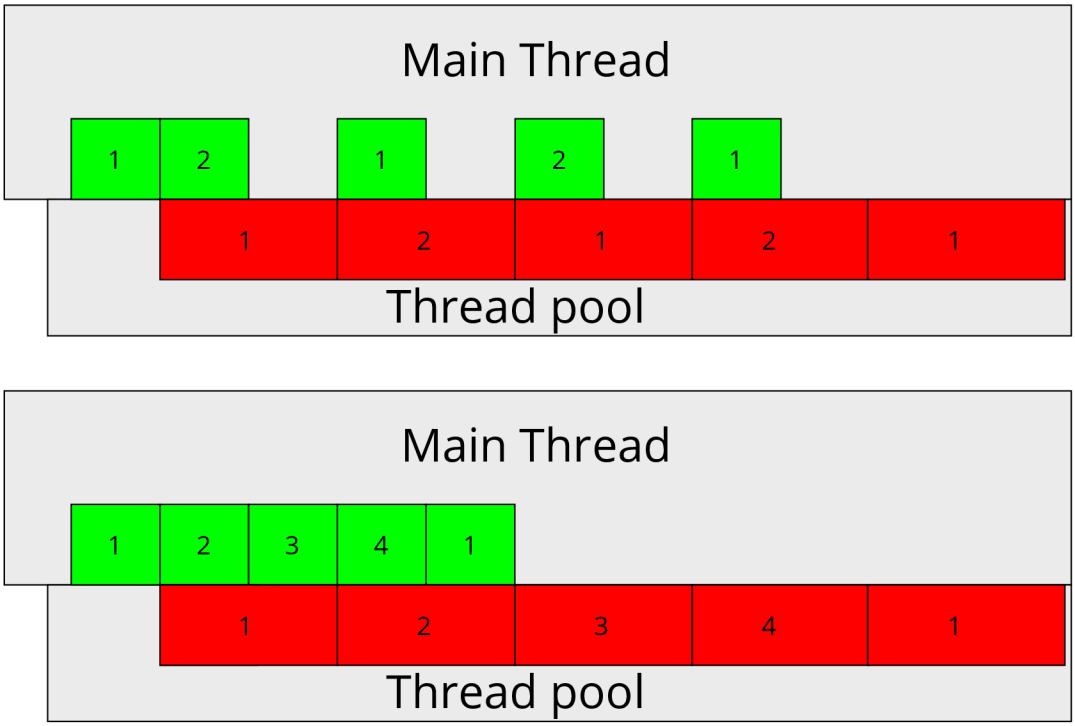
Tabulka 4.1: Srovnání výpadků cache při využití blokujícího a neblokujícího zápisu do souboru.

Přepínání kontextu - Přepnutí kontextu slouží k změně vykonávaného procesu na jádře procesoru. Před změnou běžícího procesu je nutné uložit aktuální stav procesoru, primárně tedy hodnoty registrů, na zásobník daného procesu. Při znovu zavedení běhu procesu se stav procesoru opět obnoví ze zásobníku. Režie přepnutí kontextu závisí na architektuře procesoru a také na operačním systému, avšak i přes veškeré optimalizace se jedná o náročnou operaci.

K přepnutí kontextu dochází také při volání vstupně výstupních operací. Při čekání na dokončení operace, jež trvá řádově déle než běžné aritmetické instrukce, se proces dobrovolně vzdá procesoru a je pozastaven, dokud vstupně výstupní zařízení nevyvolá přerušení, čímž informuje o dokončení vstupně výstupní operace. Při zápisu velkého množství dat pomocí knihovny HDF5 jsou data ukládána po částech (zpravidla po jednotlivých chuncích, viz podsekce 2.2.2), tedy k přepnutí kontextu dojde několikrát v rámci jednoho zápisu.

Původní implementace aplikace k-Wave s MPI využívá vždy maximálně jeden proces na každé jádro procesoru. Jelikož aplikace běžící na superpočítači jsou typicky izolované (uživatel si rezervuje celý uzel pro sebe), dochází k přepnutí kontextu minimálně. Přidáním dedikovaných vláken pro zápis nastává situace, při níž hlavní výpočetní vlákno se střídá s vláknem pro zápis o přístup k jádru procesoru.

V rámci prototypové aplikace jsem testoval dopad na výkon aplikace při vázání procesů a vláken na jednotlivé jádra mikroprocesorů. V prvním testovacím scénáři jsem navázal vlákna na stejná jádra, jako procesy, jež jej vytvořili. V druhém jsem navázal vlákna na volná jádra, jež nejsou využívání MPI procesy. Testování ukázalo, že pokud jsou vlákna pro zápis navázána na stejné jádro jako proces, jenž vlákno vytvořil, dojde k prodloužení doby běhu aplikace až o 12%.



Obrázek 4.7: Porovnání délky běhu simulace pokud je zapisující fáze delší, než fáze výpočtu. V horním schématu je naznačeno využití dvou pomocných polí pro zápis a v dolním čtyř.

Velikost vektoru polí pro ukládání dat

Během simulační fáze jsou vypočítávána data, jež mají být následně uložena do souboru, ukládána do pomocného pole. Aby se předešlo zbytečnému kopírování dat, je dané pole předáno vedlejšímu vláknu pro zápis a následující simulační krok využije jiné pole, čímž se zároveň zamezí nechtěnému přepsání dat. V běžných případech trvá zápis dat kratší dobu, než simulační krok, jak je znázorněno na obrázku 4.5. V takovém případě stačí pouze rotovat dvě pole, jelikož jedno z polí je vždy volné, vždy když začíná nový simulační krok.

V určitých případech však zápis může trvat delší dobu, než simulační krok. Například pokud je zkoumaná subdoména příliš velká a nebo je dokonce ukládána celá doména. V tomto případě se nabízí otázka, jestli není žádoucí využití většího počtu pomocných polí. V rámci projektu k-Wave jsou v každém iteračním kroku ukládána data o stejné velikosti a tedy zápis trvá přibližně stejnou dobu (v závislosti na vytížení systému). Jak je znázorněno na obrázcích 4.7, využití více pomocných polí sice zrychlí samotný výpočet, ale celkový čas běhu aplikace se nezmění. Přidaná datová náročnost tedy nepřinese v celkovém důsledku žádný užitek a proto jsem se rozhodl využít pouze dvě pomocná pole.

4.5 Postup simulace

V následujících bodech je popsán postup simulace v prototypové aplikaci se zaměřením na úkony spojené s paralelním zápisem do souboru.

Přípravná fáze

1. Hlavní vlákno vytvoří vektor polí pro zapisovaná data.
2. Hlavní vlákno vytvoří instanci `ThreadPool` s jedním vláknem pro obsluhu fronty.

Simulační fáze

1. Hlavní vlákno si podá požadavek na rezervaci pole z vektoru polí pro zápis.
2. Pokud předchozí požadavek na zápis využívající dané pole nebyl ještě vyřízen, hlavní vlákno čeká, až bude pole volné. V opačném případě by došlo k přepsání předchozích dat a nekonzistenci dat v souboru.
3. Když je pole volné, hlavní vlákno provede simulační krok v závislosti na zvoleném scénáři.
4. Po skončení simulačního kroku vloží požadavek na zápis do fronty, čímž se uzamkne použité pole.
5. Hlavní vlákno pokračuje na další simulační krok.

Ukončující fáze

1. Hlavní vlákno počká na ukončení všech zápisů pomocí volání `ThreadPool::waitEmpty`, jež zajistí, že fronta pro zápis je prázdná.
2. Hlavní vlákno zavře instanci `ThreadPool` a tím ukončí vlákno pro obsluhu fronty.
3. Hlavní vlákno zavře soubor.

4.6 Testování

Prototypová aplikace byla testována na superpočítači Salomon v Ostravě pod záštitou projektu Open Access Grant Competition. [21] Testovací parametry byly zvoleny na základě požadavků aplikace k-Wave za účelem napodobení následného reálného využití neblokujícího zápisu v této aplikaci. V první fázi byla prototypová aplikace testována s využitím jednotek uzlů a MPI procesů za účelem verifikovat základní principy využívané v této diplomové práci. Vzhledem k nízkému výpočetnímu výkonu jsem zvolil vstupní doménu o velikosti pouze 256^3 bodů mřížky, z níž v každém z 128 iteračních kroků byla ukládána data do souboru formátu HDF5. V druhé fázi probíhalo testování na větším množství uzlů na vstupní doméně o velikosti 2048^3 bodů mřížky, jež odpovídá typické velikosti vstupní domény pro MPI verzi aplikaci k-Wave.

Hlavním předmětem testování byla celková doba běhu programu a také jednotlivých dílčích částí simulace při rostoucím počtu spolupracujících MPI procesů. Pro testování jsem zvolil počet procesů odpovídající mocninám dvou, jelikož aplikace k-Wave dosahuje nejlepších výsledků právě s tímto počtem procesů. Jednotlivé uzly na superpočítači Salomon mají dohromady 24 jader na dvou fyzických mikroprocesorech, což neodpovídá mocnině dvou a tedy nevyužitá jádra lze využít pro zapisující vlákna.

Verifikace výstupního souboru

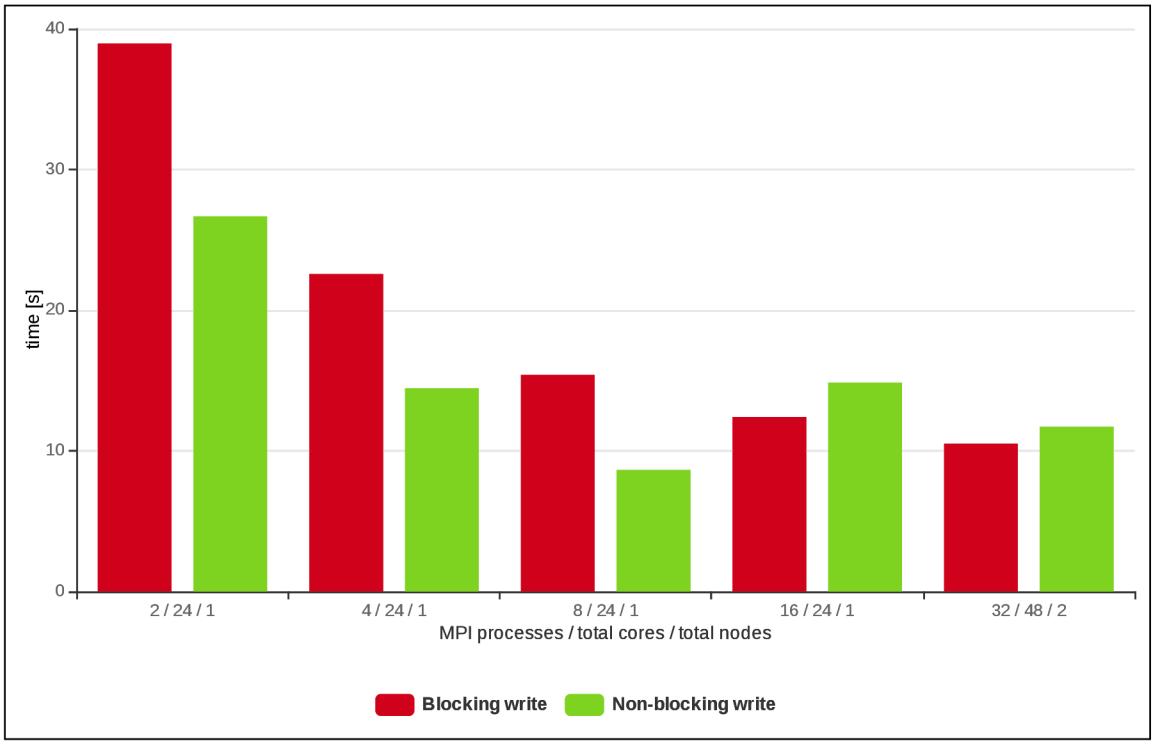
Prototypová aplikace byla navržena tak, aby výstupní dataset v HDF5 souboru byl vždy obsahově identický bez ohledu na počet využitých MPI procesů, architektury a zvolenou MPI knihovnu. Díky tomu bylo možné k verifikaci korektního zápisu dat využít aplikaci `h5diff`, jež umožňuje porovnat obsah datasetu dvou souborů formátu HDF5. Na rozdíl od Unixového nástroje `diff`, `h5diff` je schopný porovnávat pouze uložená data v datasetu a ne soubor jako celek. Ten se totiž podle standardu HDF5 může lišit na různých architekturách za účelem nejlepší optimalizace.

Nejprve jsem tedy v sériové verzi prototypové aplikace vygeneroval referenční soubor, jenž byl následně používán pro porovnávání s ostatními testovacími běhy. K dodatečné verifikaci referenčního souboru jsem využil Python skript s knihovnou *h5py*, čímž jsem ověřil správnost dat.

Měření rychlosti zápisu

Naměřený výkon zápisu prototypové aplikace v testovacích scénářích poukázal na limitace současné implementace HDF5 knihovny, jež nebyla navržena na využití více procesového přístupu zároveň s více vlákny. Běžné vstupně výstupní operace v Unixových systémech jsou implementovány jako neblokující z pohledu procesoru. Tedy při zahájení vstupně výstupní operace se proces vzdá procesoru a čeká na přerušení signalizující konec operace. Během této doby je možné využít procesor jiným procesem. Knihovna HDF5 však tento standardní přístup neimplementuje a během provádění vstupně výstupních operací si stále drží procesor.

Z tohoto důvodu také explicitní mapování dvou a více vláken na jedno jádro procesoru má velmi negativní vliv na výkon. Při kolektivních vstupně výstupních operací využívá knihovna HDF5 volání protokolu MPI k optimálnímu přeskládání dat před samotným zápisem. Avšak pokud jedno vlákno čeká na přijetí dat z jiného vlákna a neuvolní při tom

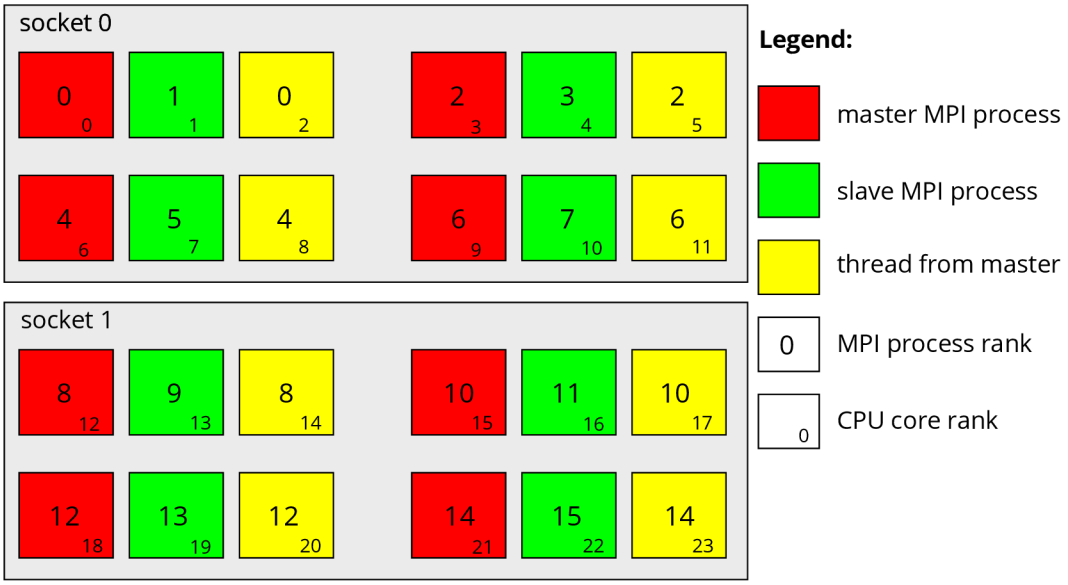


Obrázek 4.8: Porovnání celkové doby běhu prototypové aplikace s různým počtem MPI procesů při použití blokujícího a neblokujícího zápisu.

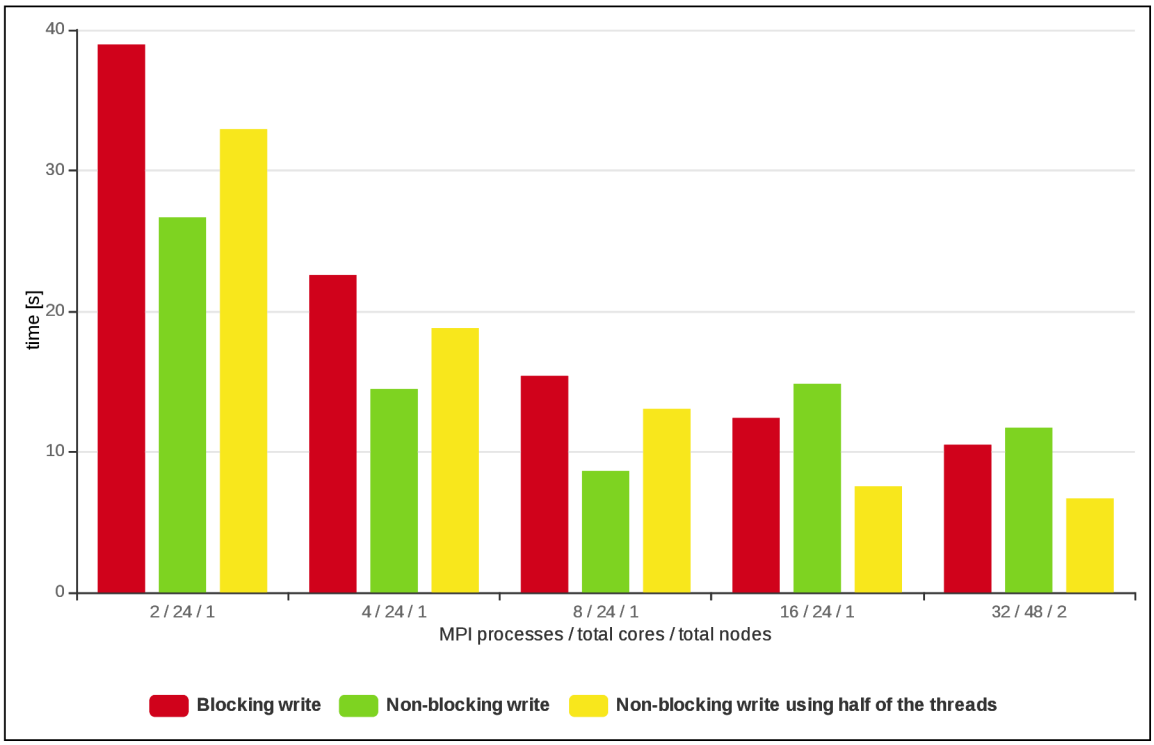
jádro procesoru, nemůže nikdy data obdržet, jelikož odesílající vlákno nemůže pokračovat, dokud operační systém neodebere procesor vláknu. Při malém počtu MPI procesů, a tedy i dedikovaných vláken na zápis, se doba běhu prodlouží až o tisíce procent. Při velkém počtu MPI procesů běh programu neskončí v rámci přiděleného časového úseku pro rezervaci uzlu.

V grafu 4.8 je znázorněn čas běhu prototypové aplikace v závislosti na počtu využitých procesů. Testování ukázalo, že neblokující zápis z pohledu aplikace byl implementován úspěšně a dochází k překrytí výpočtu v jednotlivých iteracích s neblokujícím zápisem, ale pouze v případech, dokud vlákna na neblokující zápis mají každá k dispozici volný procesor. Při využití 16 MPI procesů není již na uzlu s 24 procesory dostatečné množství procesorů pro zapisující vlákna a doba běhu aplikace se prodlouží. Tento přístup je však nevhodný, jelikož při vyšším počtu volných jader je lepší využít rovnou více MPI procesů a zapisovat blokujícím způsobem.

Pro účely projektu k-Wave na superpočítači Salomon lze však dosáhnout částečné optimalizace. Nejvyšší vhodný počet MPI procesů na jeden uzel je 16 a uzel má 24 procesorů, je tedy možné využít zbývajících 8 procesorů pro vlákna sloužící k zápisu dat. Jelikož není možné namapovat více zapisujících vláken na jedno jádro, implementoval jsem předzpracování dat, kdy si procesy po dvojích přepošlou data tak, aby byla data na zápis pouze na sudých procesech. Pomocí volání `MPI_Comm_split` se vytvoří nový komunikátor pouze pro sudé procesy. Díky tomu pouze polovina procesů se bude účastnit zápisu do souboru a tedy pouze tyto procesy vytváří dedikovaná vlákna na zápis, čímž se naplno využije všech 24 jader.

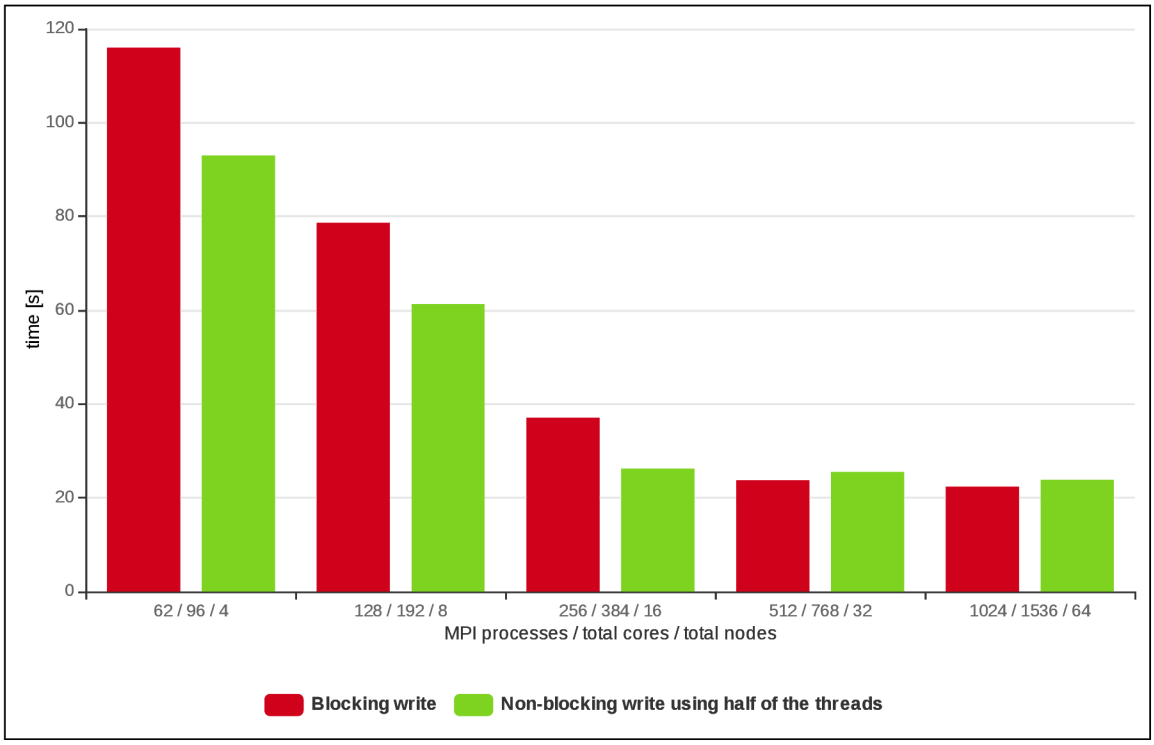


Obrázek 4.9: Optimální mapování MPI procesů a vláken na jednotlivé jádra mikroprocesoru na uzlu superpočítače Salomon.



Obrázek 4.10: Porovnání celkové doby běhu prototypové aplikace s různým počtem MPI procesů při použití blokujícího zápisu, neblokujícího zápisu a neblokujícího zápisu s přeuspořádání dat mezi procesy.

Ve výchozím nastavení operační systém nezvolí nejoptimálnější rozložení procesů a vláken na jednotlivé procesory. Je tedy nutné namapovat procesy a vlákna explicitně. Mapování procesů závisí na zvolené MPI knihovně. Například Intel MPI, jež jsem využil pro testování, umožňuje namapovat procesy pomocí proměnné prostředí `I_MPI_PIN_PROCESSOR_LIST`. Vlákna je následně možné namapovat přímo v programu pomocí volání funkce `pthread_setaffinity_np` z knihovny `pthread.h`. Příklad mapování 16 procesů a 8 vláken na jednotlivé procesory uzlu na superpočítači Salomon je znázorněno na obrázku 4.9. V grafu 4.10 je znázorněno porovnání výkonu blokujícího zápisu, neblokujícího zápisu a neblokujícího zápis s přeposíláním dat. Z testování vyplývá, že přeposílání dat přináší velkou režii a není univerzálním řešením pro libovolný počet MPI procesů, avšak dosahuje nejlepších výsledků pro plné využití celého uzlu na superpočítači Salomon.



Obrázek 4.11: Porovnání celkové doby běhu prototypové aplikace na různém počtu uzlů na doméně o velikosti mřížky 2048^3 při použití blokujícího zápisu a neblokujícího zápisu s přeuspořádání dat mezi procesy.

Druhá fáze testování probíhala již na vyšším počtu výpočetních uzlů, jenž se pohyboval od 4 (64 MPI procesů) až po 64 (1024 MPI procesů), a větší vstupní doméně o velikosti 2048^3 bodů mřížky. Při testování jsem porovnával výpočetní čas prototypové aplikace s využitím blokujícího zápisu a neblokujícího zápisu z poloviny procesů. Neblokující zápis ze všech procesů byl vypuštěn z testování, jelikož doba běhu aplikace byla vždy vyšší než u varianty s blokujícím zápisem. Jak je vidět na grafu 4.11, s využitím neblokujícího zápisu došlo ke zkrácení doby běhu prototypové aplikace průměrně o 25%, pokud byl počet MPI procesů nižší než 512. Vzhledem k velikosti vstupní domény a složitosti výpočtu je tento počet procesů již příliš vysoký, a tedy přidáváním dalších MPI procesů se již nedosahuje téměř žádného zrychlení. V takovém případě se naopak projeví režie přidaná s přepínáním kon-

textu a správy vláken, jež způsobila zpomalení prototypové aplikace s neblokujícím zápisem průměrně o 8% vůči variantě s blokujícím zápisem.

Alternativním přístupem pro obsluhu dedikovaných vláken pro zápis je využití Hyper-Threading Technology, popsané v sekci 3.2. Přidané logické procesory nemají dostatečný výkon na to, aby bylo vhodnější je využít pro více MPI procesů, jelikož by logická jádra soupeřily o fyzické prostředky procesoru. Bohužel v současné době superpočítač Salomon má tuto funkcionalitu deaktivovanou, jelikož Hyper-Threading Technology není vždy žádoucí a podpora pouze na některých uzlech by způsobila nehomogenní prostředí. Podle vyjádření technické podpory it4i, spravující tento superpočítač, se nepočítá s opětovnou aktivací Hyper-Threadingu a tedy nebylo možné tento přístup otestovat.

Kapitola 5

k-Wave

Projekt k-Wave je volně dostupná sada nástrojů, implementovaných v jazycích Matlab a C++, sloužící k simulaci šíření akustických vln v jedno, dvou a tří dimenzionálním prostoru v čase. Tato sada nástrojů má velkou řadu užití, avšak jádro celého projektu je pokročilý numerický model, využívající k-prostorovou pseudo-spektrální metodu pro řešení diferenciálních rovnic, jež popisuje lineární i nelineární šíření ultrazvukových vln v nehomogenním médiu a absorpci. Hlavní využití k-Wave se uplatňuje v medicíně při plánování operací pomocí zaostřeného ultrazvuku (focused ultrasound surgery) na určení množství energie absorbované jednotlivými částmi tkáně. [6]

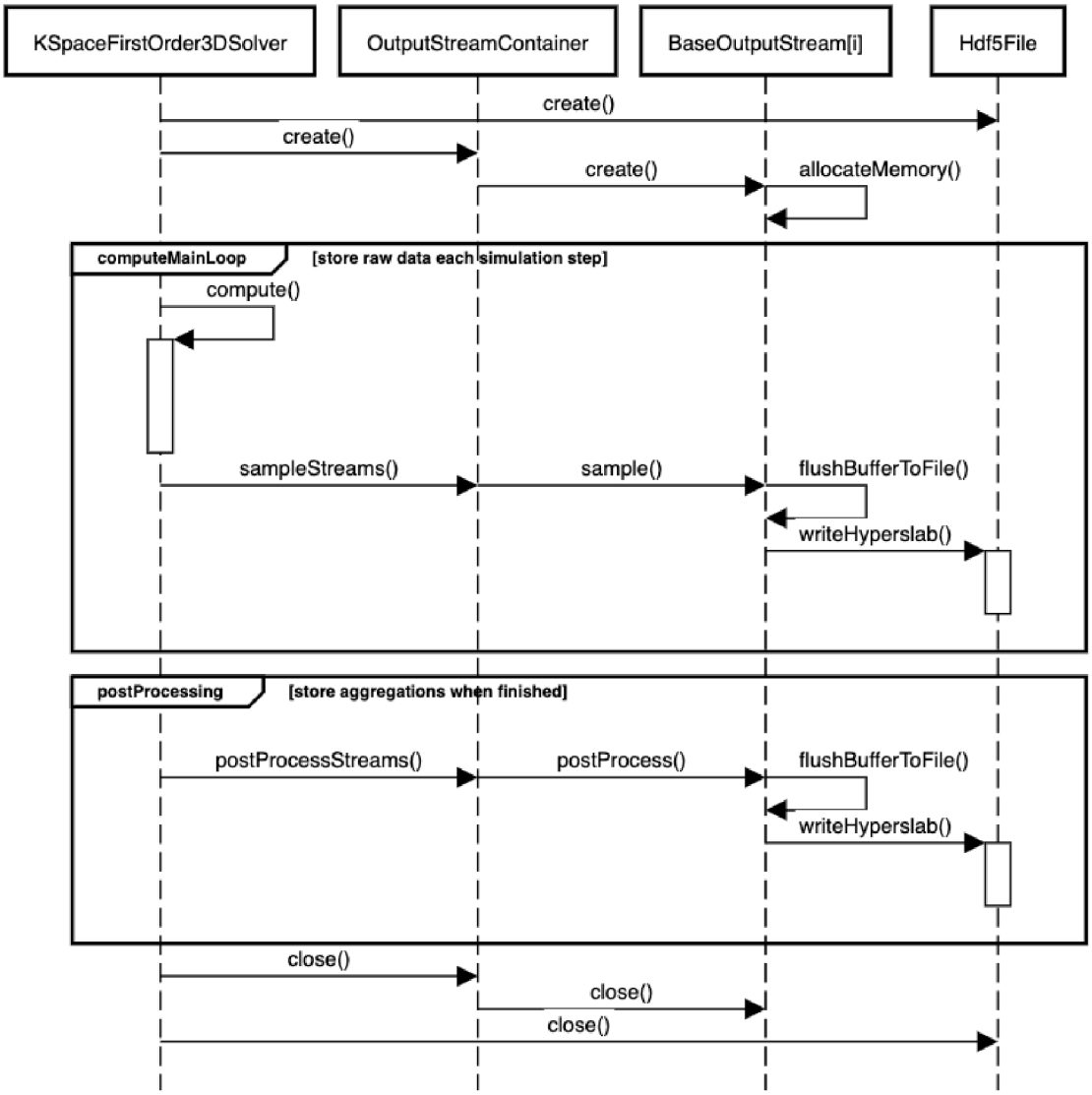
Jednotlivé části projektu k-Wave se stále vyvíjí. V současné době existuje několik implementací pro simulace různě velkých domén. Původní implementace v jazyce Matlab slouží především na simulace malých domén. OpenMP verze implementovaná v jazyce C++, využívající vektorové instrukce procesoru, umožňuje efektivnější výpočet a tedy i simulaci větších domén. Pro domény o velikosti 1024^3 a větších je již vhodné použít verzi s knihovnou MPI, jež je předmětem této diplomové práce, která umožňuje spuštění simulace na více uzlech.

5.1 Architektura

Architektura celé aplikace k-Wave ve verzi MPI je velmi komplexní. V následujícím výčtu se proto zaměřím pouze na soubory a třídy, jež jsou spojené se zápisem dat a konfigurací běhu simulace. Podrobněji budou rozebrány ty, jež byly modifikovány v rámci této diplomové práce. Konkrétní změny v implementaci jsou rozebrány v následující sekci 5.2.

Parametry simulace jsou zadány souběžně přes parametry příkazové řádky programu a vstupní soubor formátu HDF5. Parametry příkazové řádky programu slouží k definování vstupního a výstupního souboru formátu HDF5 a zkoumaných hodnot v subdoménách, jako je například rychlost šíření ultrazvuku a akustický tlak. Vstupní soubor následně definuje vlastnosti domény, polohu zkoumaných subdomén a délku simulace.

V projektu k-Wave jsou parametry programu zpracovány a předávány pomocí třídy `Parameters`, jež je implementována v souboru `Parameters/Parameters.h` jako jedináček (angl. *singleton*). Existuje tedy pouze jediná instance třídy, jež drží všechny parametry a jež je dostupná za všech ostatních tříd v projektu k-Wave. Díky tomu není nutné předávat velké množství parametrů mezi jednotlivými volání, ale parametry jsou globálně dostupné.



Obrázek 5.1: Sekvenční diagram průběhu simulace se zaměřením na zápis dat v jednotlivých iteračních krocích simulace popisující komunikaci mezi jednotlivými třídami.

Hlavní běh simulace je implementován pomocí třídy `KSpaceFirstOrder3DSolver`. Nejprve se v konstruktoru této třídy vytvoří všechny pomocné třídy pro správu domény a subdomén, jež jsou sepsány níže, a také objekty pro měření výkonu dílčích částí simulace a celkového času simulace. Všechny tyto objekty jsou následně uvolněny v destruktoru této třídy.

Když jsou všechny pomocné objekty vytvořeny, začne hlavní běh simulace voláním metody `computeMainLoop`. Počet iterací je primárně definován parametry simulace. Avšak lze také definovat maximální doba simulace, jež je zadána jako parametr programu, a simulace se ukončí předčasně. Následně je také možné pokračovat v již rozběhlé simulaci od posledního časového kroku, kde simulace skončila.

Třída `KSpaceFirstOrder3DSolver` implementuje desítky metod pro výpočet šíření ultrazvuku v doméně v jednotlivých iteračních krocích, jež jsou invokovány v závislosti na zvo-

lených parametrech simulace. Tato diplomová práce se zabývá především vzorkováním rychlosti šíření ultrazvuku a akustického tlaku v jednotlivých iteračních krocích, jelikož v tomto nastavení se ukládají data po každém iteračním kroku a ukládá se velké množství dat. Ostatní vlastnosti, jako je například maximální nebo minimální akustický tlak, jež se ukládají do výstupního souboru pouze na konci celé simulace, není nutné optimalizovat, jelikož nemůže dojít k překrytí zápisu a výpočtu. V rámci této diplomové práce byla však nutná modifikace i pro tyto případy z důvodů limitací knihovny HDF5, aby nedošlo k nekorektnímu chování programu. Velmi abstraktní průběh simulace je znázorněn sekvenčním diagramem na obrázku 5.1, jenž se zaměřuje pouze na průběh zápisu dat v jednotlivých iteračních krocích.

Správa simulačních dat je implementována ve třídě `MatrixContainer`, jež zpřístupňuje jednotlivé složky vstupní domény. Doména je uložená jako struktura polí, přičemž každá vlastnost domény je reprezentována maticí. V závislosti na zkoumané vlastnosti rozdělujeme matice na dva základní typy podle způsobu distribuce: `broadcasted` matice jsou celé dostupné každému z MPI procesů a `scattered` matice jsou rozděleny mezi jednotlivé MPI procesy, přičemž každý proces má k dispozici pouze dílčí část domény. Vstupní doména je načtena ze souboru pouze jednou před začátkem simulace a v každém iteračním kroku editována. Díky tomu je paměťová náročnost simulace známá již před samotným spuštěním.

V současné implementaci aplikace k-Wave jsou data předávána primárně ukazatelem, aby nedocházelo ke zbytečnému kopírování dat. Pokud však chceme implementovat neblokující zápis do souboru, musíme ochránit data před přepsáním, než bude zápis dokončen. Způsob implementace bude popsán v následující sekci 5.2.

Správa výstupních proudů je implementována ve třídě `OutputStreamContainer` a slouží jako jednotné rozhraní pro veškeré zápisy subdomén do výstupního souboru. Pro každou zkoumanou vlastnost, jež bude zapsána do souboru, se vytvoří instance jedné z podtříd implementující rozhraní rodičovské třídy `BaseOutputStream`. Konkrétní implementace závisí na zvolených parametrech simulace a tvaru subdomény.

První podtřídou je `IndexOutputStream` sloužící k uložení senzorové masky libovolného tvaru. Jednotlivé body jsou při uložení serializovany do jednorozměrného pole, jež je uloženo do výstupního souboru. `CuboidOutputStream` umožňuje uložit subdomény tvaru kvádru. V rámci jedné instance této třídy je možné uložit více subdomén současně, přičemž ve výstupním souboru je každá subdoména uložena ve vlastním datasetu a má tvar trojrozměrného pole. Poslední podtřídou je `WholeDomainOutputStream` umožňující uložení celé domény do výstupního souboru.

Velmi důležitým parametrem všech výstupních proudů je redukční operátor. Pokud není žádný redukční operátor použit (hodnota `kNone`), ukládají se surová naměřená data v každém iteračním kroku. V případě subdomén tvaru kvádru má výstupní dataset podobu čtyřrozměrného pole, přičemž první dimenze reprezentuje časovou složku. Obdobně při použití senzorové masky má výstupní dataset podobu dvourozměrného pole. Bez redukčního operátoru není nutné udržovat žádná data v instancích implementující výstupní proud mezi jednotlivými simulačními kroky, jelikož v každém iteračním kroku jsou aktuální data dostupná přes ukazatele na výpočetní matice.

Dalšími hodnotami pro redukční operátor jsou maximum, minimum a kvadratický průměr. V těchto případech si instance výstupních proudů drží mezivýsledky po celou dobu simulace a agregovaná data jsou zapsána do výstupního souboru jednorázově až po skončení hlavního běhu simulace.

Mějme například zadání simulace o 500 časových krocích, jež se zaměřuje na surová data o rychlosti šíření ultrazvuku a akustickém tlaku v jednotlivých krocích simulace a také na maximální hodnoty akustického tlaku v průběhu celé simulace. Simulace zkoumá pět subdomén tvaru kvádru. Vytvoří se tedy tři instance třídy `CuboidOutputStream`, jedna pro každou zkoumanou vlastnost. Dvě instance budou zapisovat data každý simulační krok a provedou dohromady 5 000 zápisů (2 zkoumané vlastnosti, krát 500 simulačních kroků, krát 5 subdomén). Třetí instance provede pouze 5 zápisů agregovaných dat, jednu pro každou subdoménu. Celkem tedy během této simulace bude provedeno 5 005 kolektivních zápisů, jež v původní implementaci k-Wave jsou všechny blokující pro všechny MPI procesy.

Vstupně výstupní operace jsou zastřešeny ve třídě `Hdf5File`. V aplikaci k-Wave se využívají dvě instance, jedna pro vstupní soubor a druhá pro výstupní. Tato třída poskytuje abstrakce nad veškerou prací s knihovnou HDF5 a poskytuje jednoduché rozhraní pro kolektivní operace.

5.2 Implementace neblokujícího zápisu

K implementaci neblokujícího zápisu v aplikaci k-Wave jsem využil poznatky a konstrukce z prototypové aplikace. Ke správě vstupně výstupních operací jsem využil opět knihovnu *ThreadPool* [26] využívající perzistentní vlákna na obsluhující frontu požadavků, jež byla podrobněji popsána v sekci 4.4. Každý MPI proces vytváří v konstruktoru třídy `KSpaceFirstOrder3DSolver` jednu instanci třídy `ThreadPool`, jež je dále předávána jako reference do správce výstupních proudů `OutputStreamContainer` a dále do jednotlivých instancí výstupních proudů. Díky tomu veškeré zápisy do výstupního souboru jsou serializovaný přes jednu frontu a nemůže dojít ke kolizi souběžných zápisů, což by v současné implementaci knihovny HDF5 způsobilo chybu programu, jak bylo popsáno v sekci 4.2.

Následně jsem modifikoval jednotlivé podtřídy výstupních proudů vycházející ze třídy `BaseOutputStream`. Prvním krokem byla správa paměti, jelikož data určená na asynchronní zápis musí být chráněna před přepsáním. V každé podtřídě je reimplementovaná metoda `BaseOutputStream::allocateMemory` tak, aby se alokovalo větší množství polí pro zapisovaná data. Jak bylo zjištěno z testování na prototypové aplikaci, zpravidla stačí pouze dvě rotující pole, nicméně množství rotujících polí je možné modifikovat přes parametry podtřídy. To ovšem neplatí pro výstupní proudy využívající redukční funkce, jež provádí pouze jeden zápis na konci simulace, a tedy není nutné zbytečně alokovat větší množství paměti. Ukazatel `mStoreBuffer`, jenž v původní implementaci ukazoval na jediné pole dat pro zápis, se nyní v každém iteračním kroku mění tak, aby ukazoval na aktuální pole z rotujícího seznamu polí. Díky tomu nebyl nutný větší zásah do kódu a reimplementace zděděných metod ze třídy `BaseOutputStream`, jelikož rozhraní zůstalo zachováno. Obdobným způsobem byla pozměněna metoda `freeMemory`, aby uvolnila všechna pole pro zapisovaná data.

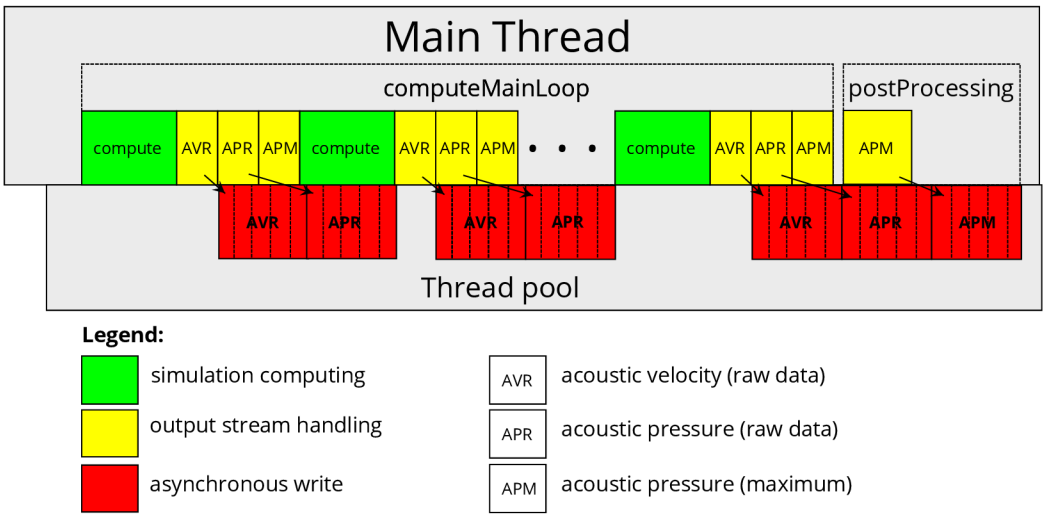
Zápis do souboru se v jednotlivých iteračních krocích simulace provádí v metodě `sample`, pokud nebyla použita žádná redukční operace. Pokud ano, pouze se aktualizují data ve výstupním proudu a simulace pokračuje dále bez zápisu do souboru. V ukázce 5.1 je znázorněn pseudokód metody `sample`, jenž popisuje průběh vložení požadavku na asynchronní zápis do fronty. Obdobně jako v prototypové aplikaci se využívá vektor polí na zápis dat a stejně velký vektor odpovídajících instancí třídy `std::future`. Nejprve se zkontroluje, že žádný z předcházejících asynchronních operací již nevyužívá aktuálně vybrané pole voláním `std::future::wait`. Když je pole volné, překopírují se do pole data z aktuálního

simulačního kroku a vloží se požadavek na zápis do fronty. Návratovou hodnotou z této operace je opět instance `std::future`, jež se uloží, aby později nemohlo dojít k nechtěnému přepsání dat v následujících iteracích simulace. Následně se inkrementuje index aktuálního pole pro další iteraci a také se aktualizuje ukazatel `mStoreBuffer`.

```
1  # wait for last write using current buffer to finish
2  await futures[currentBufferIndex]
3  # move data to buffer
4  buffers[currentBufferIndex] = sampledData
5  # put request to write data to queue and save future object
6  futures[currentBufferIndex] = enqueue(write, buffers[currentBufferIndex])
7  # update current buffer index
8  currentBufferIndex = (currentBufferIndex + 1) % totalCountOfBuffers
9  # update pointer to current buffer in order to perserve compatibility with inherited methods
10 currentBufferPointer = buffers[currentBufferIndex]
```

Ukázka 5.1: Pseudokód metody `sample` pro výběr a rezervaci pole na zapisující data a provedení zápisu.

V případě použití redukčního operátoru jsou požadavky také vkládány do fronty, aby nedocházelo ke kolizím se zápisy z jiných výstupních proudů. Pokud by se prováděly zápisy zároveň, došlo by ke chybě programu kvůli limitacím knihovny HDF5. Na konci simulační fáze se čeká na vyprázdnění fronty požadavků na zápis, aby celkový čas simulace odpovídal realitě, ale také hlavně aby všechna data ve výstupním souboru byla konzistentní.

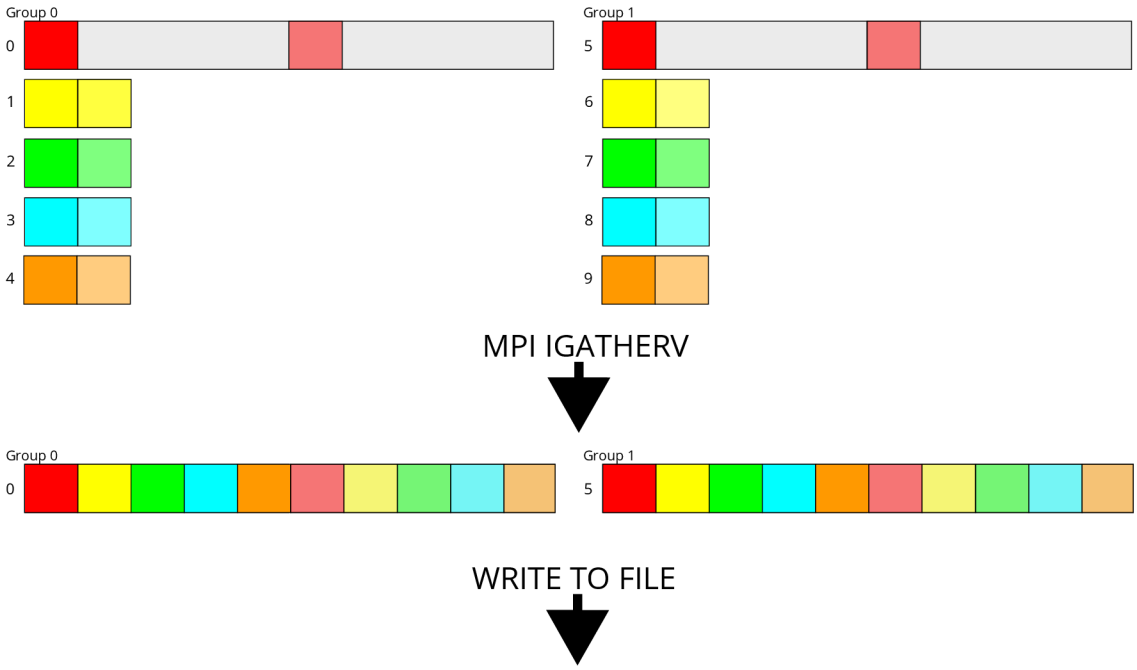


Obrázek 5.2: Průběh simulace zkoumající průběžný akustický tlak, průběžnou rychlost šíření ultrazvuku a maximální akustický tlak v pěti subdoménách.

Na obrázku 5.2 je opět využit příklad z předcházející sekce 5.1. V pěti subdoménách jsou zkoumány jsou tři vlastnosti: rychlost kmitání částic v každém kroku, akustický tlak v každém kroku a maximální akustický tlak během celé simulace. Diagram se zaměřuje především na serializaci zápisů do fronty požadavků a zápis s využitím redukčního operátoru po skončení hlavní simulační fáze.

Přeuspořádání dat

Testování na prototypové aplikaci a následně také na upravené verzi aplikace k-Wave ukázalo, že není vhodné zapisovat z pomocných vláken všemi procesy. Implementoval jsem tedy druhou verzi, taktéž využívající vlákna pro neblokující zápis, jež nejprve přeuspořádá data pouze do části procesů a následně je zápis prováděn menším počtem vláken pouze z těchto řídících procesů.

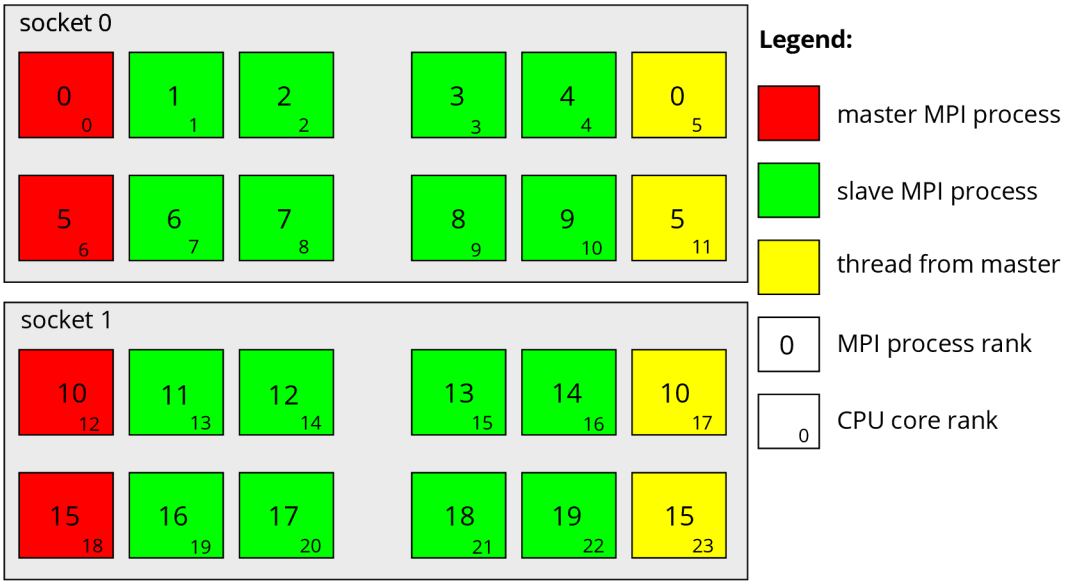


Obrázek 5.3: Průběh zápisu dvou subdomén tvaru kuboidu v jednom simulačním kroku, přičemž data jsou sesbírána ze všech MPI procesů v rámci skupiny do řídícího procesu pomocí volání `MPI_Igatherv` a následně zapsány do souboru pouze ze řídících procesů.

Jak je znázorněno v obrázku 5.3, řídící procesy si v úvodní fázi pomocí volání `MPI_Gather` sesbírají velikosti pomocných polí pro zápis ze všech procesů ve své skupině a alokují si dostatečný prostor, aby si mohli tyto data uložit. V každém iteračním kroku simulace si sesbírají data, jež mají být zapsána do souboru voláním `MPI_Igatherv`. Variabilní funkce `MPI_Igatherv` se využívá kvůli procesům, jež mají přiřazené okraje senzorové masky a tedy mají různou velikost zapisovaných dat. Následný zápis pak provádí pouze vlákna z řídících procesů a tedy počet vláken pro zápis je výrazně nižší podle toho, jak velké jsou zvolené skupiny. Na obrázku 5.4 je znázorněno rozložení procesů a vláken na jednom uzlu superpočítače Salomon, jenž má k dispozici dva mikroprocesory o celkovém počtu 24 jader, se zvolenou velikostí skupiny o pěti procesech.

5.3 Testování

Aplikace k-Wave ve verzi MPI byla obdobně jako prototypová aplikace testována na superpočítači Salomon v Ostravě pod záštitou projektu Open Access Grant Competition. [21] Pro testování jsem zvolil vstupní domény o velikostech 512^3 a 1024^3 bodů mřížky, což odpovídá běžnému využití MPI verze aplikace k-Wave v produkčním prostředí. Pro verifikaci



Obrázek 5.4: Rozložení MPI procesů a vláken na uzlu superpočítače Salomon s využitím řídících procesů a velikosti skupiny o pěti procesech.

výstupního souboru i porovnání celkového běhu aplikace jsem využil současnou implementaci aplikace k-Wave.

Verifikace výsledků

Data ve výstupním souboru nelze verifikovat tak jednoduše, jak tomu je u prototypové aplikace. Výstupní data aplikace k-Wave mají většinou datový formát čísel s pohyblivou řádovou čárkou (angl. float) a může tedy jednoduše dojít k zaokrouhlovací chybě. Pro výpočet jednotlivých simulačních kroků se také využívá distribuovaného výpočtu pro diskrétní Fourierovou transformaci pomocí knihovny Fastest Fourier Transform in the West (FFTW). [12] Vzhledem k povaze výpočtu dochází k různým chybám zaokrouhlení i při opětovném spuštění aplikace k-Wave se shodnou konfigurací a na shodné architektuře.

V této diplomové práce nebyl modifikován samotný výpočet simulace, ale pouze způsob zápisu do výstupního souboru. I v tomto případě mohlo dojít potencionální chybou v programu k inkonzistenci dat ve výstupním souboru následujícími způsoby:

- Požadavek na zápis ve frontě nebyl zpracován a data nebyla uložena vůbec.
- Pole dat pro zápis bylo přepsáno v některém z následujících simulačních kroků a došlo k nekonzistenci dat.
- Index hyperslabu byl přepsán v některém z následujících simulačních kroků a data byla uložena na špatnou pozici v datasetu.

```
$ # print help message
$ python compare.py --help
Usage: compare.py [OPTIONS] FILE1 FILE2 DATASET
  Calculate magnitude of error in datasets.
  FILE1: path to the first file
  FILE2: path to the second file
  DATASET: dataset or group path
Options:
  --threshold FLOAT threshold for difference in comparison
  --help Show this message and exit.
$
$ # compare specific dataset
$ python compare.py output.h5 reference.h5 /p/1
Over Lf threshold: 0/592
Over Linf threshold: 0/592
Maximum Lf difference: 1.111531332753657e-06
Maximum Linf difference: 1.332183842350787e-06
$
$ # compare all datasets in a group (all subdomains)
$ python compare.py output.h5 reference.h5 /p
Over Lf threshold: 0/1184
Over Linf threshold: 0/1184
Maximum Lf difference: 2.264634531456977e-06
Maximum Linf difference: 3.0725709621037822e-06
```

Ukázka 5.2: Ukázka spuštění testovacího skriptu `compare.py`.

Součástí sady nástrojů k-Wave je testovací nástroj `kWaveTester`, implementován v programovacím jazyce Matlab, jež mimo jiné umožňuje na jednom vstupním setu spustit postupně Matlabovou verzi programu a také C++ verzi programu a následně porovnat výstupní data. Tento přístup testování je vhodný především pro verifikaci samotného výpočtu simulace. Pro účely této práce, kdy se simulují velké domény paralelně stovkami procesů, by výpočet Matlabové verze programu trval velice dlouho, jelikož by výpočet Matlabové verze pro porovnání probíhal pouze na jednou uzlu.

Pro validaci výstupního souboru jsem proto v rámci této diplomové práce implementoval testovací skript v jazyce Python, jenž se nachází v souboru `Scripts/compare.py` využívající knihovny *h5py*. Tento skript umožňuje porovnat dataset ze dvou výstupních souborů a vypočítat relativní odchylku způsobenou chybami zaokrouhlováním. Díky tomu je možné pustit původní MPI verzi k-Wave a následně modifikovanou verzi k-Wave a pouze porovnat výstupní soubory, čímž se testování výrazně zrychlí oproti využití nástroje `kWaveTester`. Příklad využití tohoto testovacího nástroje je v zobrazen v ukázce 5.2.

$$\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2 \right)^{\frac{1}{2}} \tag{5.1}$$

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n} |a_{ij}| \tag{5.2}$$

Ukázka 5.5: Výpočet Frobeniové a nekonečné normy matice A o rozměrech $m \times n$.

Relativní odchylka v datasetech se v testovacím skriptu vypočítává pomocí maticové normy. Maticová norma je zobrazí, jež matici A z $T^{m \times n}$, kde T je těleso reálných čísel a m, n jsou rozměry matice, přiřadí reálné číslo $\|A\|$. Testovací skript využívá Frobeniové (5.1) a nekonečné (5.2) normy jejichž výpočet je matematicky znázorněn v ukázce 5.5. Vzhledem k povaze potenciálních chyb jsou jednotlivé časové složky porovnávány nezávisle na sobě. Každý časový krok, reprezentován trojrozměrným polem, je serializován na dvourozměrné pole a podle vzorce (5.3) je vypočítaná relativních odchylka vůči referenčnímu datasetu s využitím Frobeniové normy.

$$e_F = \max\left\{\frac{\|A-B\|_F}{\|A\|_F}, \frac{\|A-B\|_F}{\|B\|_F}\right\} \tag{5.3}$$

Ukázka 5.6: Výpočet relativních odchylky matic A a B s využitím Frobeniové normy.

Testování proběhlo úspěšně a modifikovaná verze aplikace k-Wave se s referenčním řešením shodovala vždy minimálně na pět desetinných míst, což je v normě odchylky způsobené kumulativní zaokrouhlovací chybou. Pro verifikaci jsem také porovnal výstupní soubory ze dvou nezávislých běhů současné verze aplikace k-Wave využívající blokující zápis a relativní odchylka byla totožná na počet řádů.

Překrytí paralelního zápisu

Testování překrytí paralelního zápisu byl netriviální úkol vzhledem k povaze testované aplikace a také testovacího prostředí. V současné době množství nástrojů na profilování a debuggování distribuovaných aplikací v superpočítačových prostředí je výrazně nižší než pro konvenční sériové aplikace. Mezi ty nejpokročilejší profilovací nástroje se řadí například ARM MAP, jenž je součástí sady nástrojů ARM Forge, a Score-P. Oba tyto nástroje byly podrobněji popsány v sekci 2.3. Během testování se bohužel ukázalo, že ani jeden z těchto dvou nástrojů neumožňuje profilování aplikací, jež využívají současně knihovnu MPI a více vláken na proces.

Nástroj ARM MAP při spuštění profilování vícevláknové aplikace s knihovnou MPI vypíše chybovou hlášku, že profilovací informace z vedlejších vláken jsou ignorovány. Využití tohoto nástroje pro překrytí paralelního zápisu tedy nebylo možné. Nástroj Score-P má částečnou podporu pro vlákna vytvořená pomocí knihovny `pthread.h`, avšak podpora pro knihovnu `thread`, již jsem využil v této diplomové práci, opět chybí. Score-P při profilování rozlišuje tři typy operací: výpočet, MPI volání a vstupně výstupní operace. Kvůli nedostatečné podpoře pro knihovnu `thread` je veškerá práce ve vláken považována za fázi výpočtu, což nereflektuje realitu, jelikož v této diplomové práci byla vlákna využita téměř výhradně pro vstupně výstupní operace. Výsledky profilování pomocí nástroje Score-P jsou přiloženy na paměťovém médiu A ve složce `/Misc`.

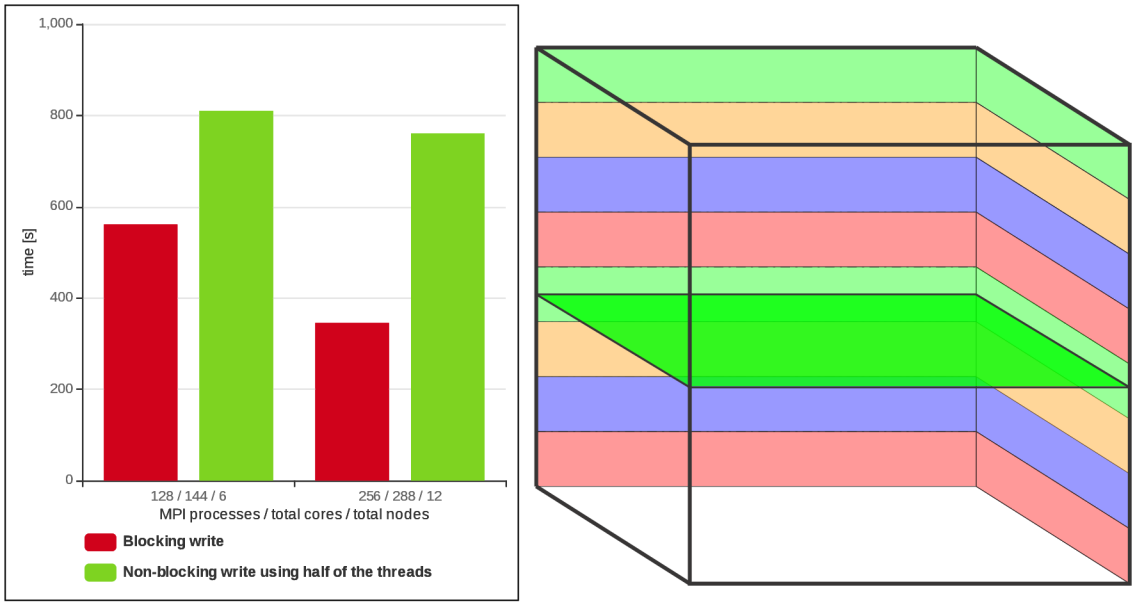
Překrytí paralelního zápisu s výpočtem simulace bylo možné tedy testovat pouze porovnáním celkového času běhu aplikace s původní implementací aplikace k-Wave. Bohužel ani v tomto případě nebylo možné naměřit exaktní výsledky vzhledem k povaze testovacího prostředí. Na superpočítači Salomon je sice možné si alokovat uzly pouze pro vlastní výpočet aplikace a tedy výpočetní čas simulace se téměř nemění, avšak všechny uzly sdílí jeden souborový systém Lustre, který má omezenou propustnost. Samotná doba čtení a zápisu do souboru je tedy velmi závislá na vytížení souborového systému ostatními uživateli. V době vypracování této diplomové práce byl superpočítač velmi vytížen. Běžným stavem je plné vytížení všech 1008 výpočetních uzlů s dalšími stovky požadavků na alokování zdrojů čekajících ve frontě. Pokud například probíhalo testování na 64 uzlech superpočítače Salomon, zbylých 944 uzlů bylo využito ostatními uživateli sdílející přístup k souborovému systému. Celková doba běhu aplikace nad stejnými vstupními daty při shodném počtu jader se v extrémních případech lišila až o desítky procent. Níže uvedená data byla proto spočítána jako průměr několika nezávislých měření.

Zápis všemi procesy

Neblokující zápis do výstupního souboru všemi procesy v aplikaci k-Wave vykazuje stejné problémy jako v prototypové aplikaci. Pokud jsou volná jádra mikroprocesoru na obsluhu dedikovaných vláken, je celková doba běhu programu nižší, než při blokující variantě. Ovšem opět je v takovém případě vhodnější využít více MPI procesů a zapisovat blokujícím způsobem. Pokud fyzická jádra nejsou volná a ani není zapnutá funkce Intel Hyperthreading dojde k prodloužení doby běhu programu až o desítky procent kvůli blokujícímu čekání implementovaného v knihovně HDF5.

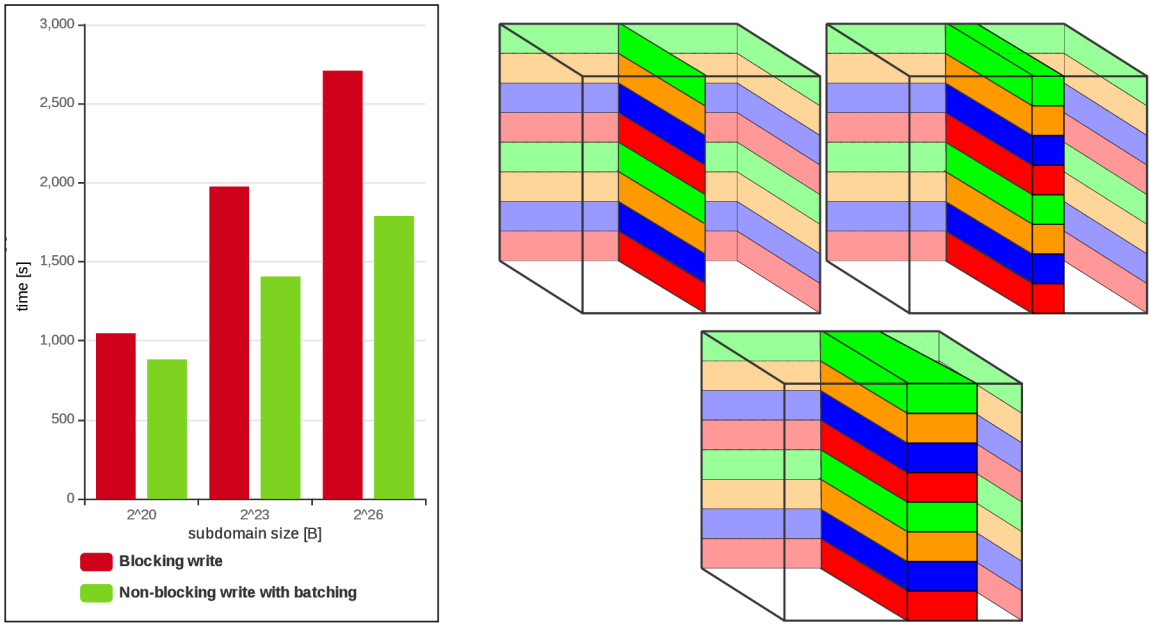
Zápis s kumulací dat

Neblokující zápis s kumulací dat sebou přináší poměrně velkou režii v závislosti na zvolené velikosti skupin procesů pro kumulaci a také na tvaru subdomény. Nejhoršího výkonu při neblokujícím zápisu s kumulací dat dochází při zvolení subdomény reprezentující transverzální řez, jak je znázorněno na obrázku 5.7. V tomto případě jsou data uložena pouze v jednom procesu, jenž pomocí volání `MPI_Igatherv` musí přesunout svá data do hlavního procesu své skupiny, který následně provede zápis. Režie přidaná přesunem dat a také správou vláken je vyšší než režie zápisu, a tedy se také výrazně prodlouží celková doba vykonávání programu.

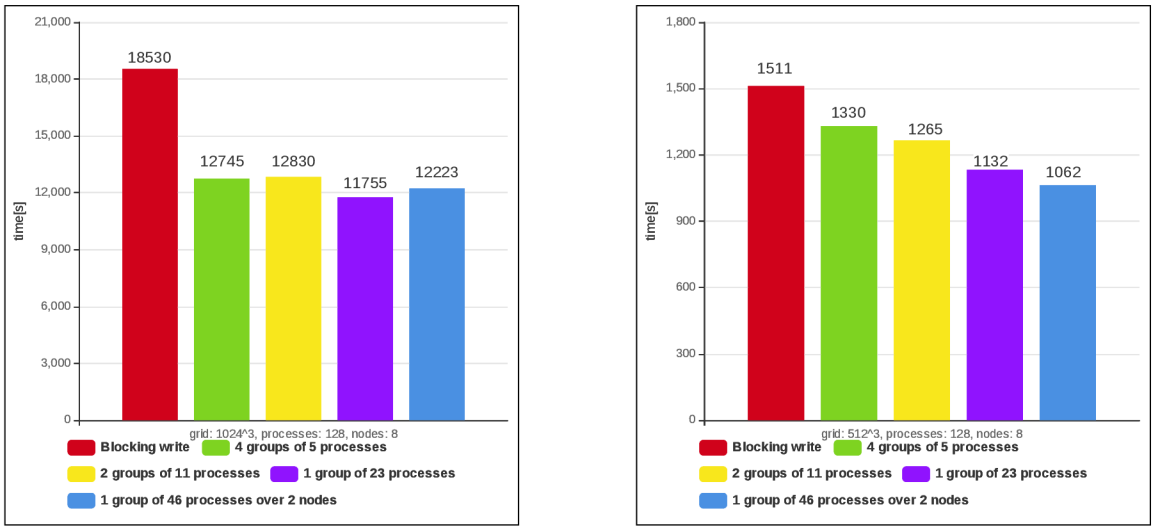


Obrázek 5.7: Testování neblokujícího zápisu s kumulací dat a výběrem subdomény reprezentující transverzální řez. Velikost vstupní domény byla zvolena 512^3 bodů mřížky. V každém z 1183 iteračních kroků se uložilo 1MB dat, a tedy výstupní soubor má celkovou velikost přibližně 1GB. Data byla agregována v rámci každého uzlu a vlákna dedikovaná na zápis byla mapována na volná fyzická jádra procesoru.

Největší vliv na celkovou dobu běhu programu má množství dat, jež jsou zapisována v každém iteračním kroku. Neblokující zápis je vhodné použít pouze v případě, pokud délka zápisu dat je dostatečně velká, aby překryla režii správy vláken a přeskládávání dat. V grafu na obrázku 5.8 je znázorněna celková doba běhu simulace s blokujícím a neblokují-



Obrázek 5.8: Porovnání blokujícího a neblokujícího zápisu s kumulací dat s rostoucí velikostí subdomény zasahující do všech procesů. Testování proběhlo s využitím 128 MPI procesů na 6 uzlech superpočítače Salomon. Vstupní doména byla zvolena o velikosti 512^3 bodů mřížky. V každém z 1183 iteračních kroků se uložilo 1MB, 8MB a 256MB dat, čemuž odpovídá celková velikost výsledného souboru přibližně 1GB, 8GB a 256GB.

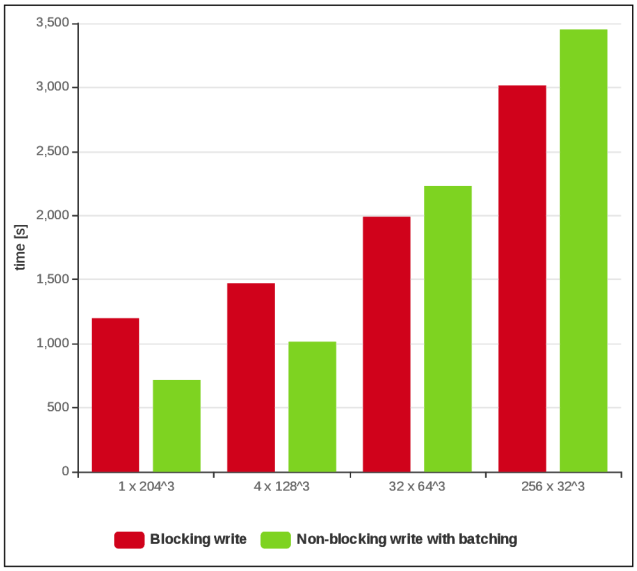


Obrázek 5.9: Porovnání neblokujícího zápisu s různě zvolenými velikostmi skupin pro kumulaci dat. Vstupní domény byly zvoleny o velikosti 1024^3 a 512^3 bodů mřížky.

cím zápisem s různě zvolenými velikostmi subdomén, jež zasahují do všech procesů. Pokud je subdoména dostatečně velká, dojde ke zrychlení doby běhu programu až o 33%.

Velikost skupin procesů pro kumulaci dat nemá zásadní vliv na celkovou dobu běhu programu dokud všechny procesy v rámci skupiny jsou na stejném uzlu a množství zapisovaných dat je dostatečně velké. Pokud je zapisováno menší množství dat je vhodnější zvolit větší velikosti skupin, jak je znázorněno na grafech 5.9. Knihovna MPI totiž provádí shlukování malých individuálních dotazů na vstupně výstupní systém, jež bylo podrobněji rozebráno v sekci 3.1 a znázorněno na obrázku 3.3, a může dojít i k přesunu dat mezi jednotlivými uzly. V takovém případě je vhodnější zvolit větší velikost skupin procesů pro kumulaci dat zasahující i přes více uzlů, čímž se sníží režie pomocných vláken a také MPI zpráv.

Jelikož aplikace k-Wave dosahuje nejlepších výsledků pokud je zvolen počet MPI procesů o velikosti mocnin dvou, je vhodné tomu také přizpůsobit velikost skupin procesů pro kumulaci dat. Máme-li například k dispozici 6 uzlů, přičemž každý má k dispozici 24 fyzických jader, je maximální množství MPI procesů vhodných pro simulaci 128. V takovém případě je vhodné zvolit skupiny pro kumulaci dat o velikosti 11 MPI procesů. Na každý uzel tedy připadne 22 MPI procesů a 2 dedikovaná vlákna pro zápis dat, čímž dojde k nejefektivnějšímu využití zdrojů.



Obrázek 5.10: Porovnání blokujícího a neblokujícího zápisu s kumulací dat s rostoucím počtem subdomén. Vstupní doména byla zvolena o velikosti 512^3 bodů mřížky. V jednotlivých testovacích scénářích se mění počet a velikost subdomén tvaru kuboidu, jež jsou následující: 1×204^3, 4×128^3, 32×64^3 a 256×32^3. Ve všech testovacích scénářích se tedy v každém z 1183 iteračních kroků uložilo stejné množství dat (8MB), pouze s jiným rozložením.

Dalším faktorem ovlivňující dobu zápisu dat je počet zkoumaných subdomén a s tím související počet datasetů ve výstupním souboru. Zatímco při použití indexové senzorové masky jsou výstupní data uložena vždy pouze v jednom datasetu, v případě senzorové masky tvaru kuboidu odpovídá v projektu k-Wave každé subdoméně jeden dataset. Kolektivní operace zápisu do HDF5 souboru umožňuje v jeden okamžik zapisovat data vždy pouze do jednoho datasetu a vzhledem limitaci současné implementace knihovny HDF5, jež byla podrobně popsána v sekci 4.2, není možné tuto operaci volat paralelně. Pokud tedy například zkoumáme akustický tlak ve 32 kuboidech, znamená to 32 kolektivních zápisu

do souboru v každém iteračním kroku. Na obrázku 5.10 je znázorněno porovnání celkové doby běhu simulace, jež v každém iteračním kroku sice ukládá objemově stejné množství dat, ale do různého počtu datasetů. V původní implementaci aplikace k-Wave je doba běhu simulace se zápisem do 256 datasetů až téměř třikrát pomalejší než se zápisem stejného objemu dat do jednoho datasetu. V případě neblokujícího zápisu je tento rozdíl ještě výraznější kvůli přidané režii kopírování dat do pomocných polí, správy vláken a přeuspořádání dat. Jelikož zápis dat v každém iteračním kroku trvá déle než samotný výpočet simulace, musí hlavní vlákno programu čekat na uvolnění pomocného pole pro zápis dat z předcházejícího kroku.

V případě velmi malých kuboidů nastává situace, že celá subdoména je uložena pouze v paměti jednoho MPI procesu, který provádí zápis, zatímco všechny ostatní procesy čekají. V takovém případě by tento proces mohl provést individuální zápis bez nutnosti synchronizace s ostatními procesy, jelikož do datasetu zapisuje jediný, a tedy zápis do jednotlivých datasetů by bylo možné paralelizovat. Pokud by takto malé datasety byly umístěny na rozmezí dvou procesů, data by si mohly přeposlat a opět by provedl individuální zápis pouze jeden z nich. Tento přístup by však vyžadoval implementovat analýzu rozložení dat a pravidla pro přeposílání dat, což by mohlo být zajímavé budoucí rozšíření této diplomové práce.

Z testování vyplynulo, že využití neblokujícího zápisu není univerzálním řešením kvůli přidané režii pro libovolné simulační scénáře, ale je vhodné využít pokud jsou splněny alespoň částečně následující podmínky:

- Velikost zapisovaných dat je dostatečně velká vůči délce výpočtu simulačního kroku. Například pro doménu o velikosti 512^3 bodů mřížky a 128 MPI procesů to odpovídá velikosti 2MB na jeden iterační krok.
- Data jsou alespoň částečně rovnoměrně rozložena mezi MPI procesy.
- Na jednotlivých uzlech jsou volná fyzická jádra, jež nemohou být využita pro MPI procesy kvůli požadavku aplikace k-Wave na množství procesů o velikosti mocnin dvou.
- Zápis není prováděn do příliš velkého počtu datasetů v každém iteračním kroku.

Kapitola 6

Závěr

Cílem této diplomové práce bylo seznámit se s principy distribuovaného vstupu a výstupu v superpočítačových aplikacích se zaměřením především na souborový systém Lustre, protokol pro synchronizaci procesů MPI a souborový formát HDF5. Následně se seznámit s požadavky aplikace k-Wave na vstupně výstupním rozhraní, navrhnout a implementovat neblokující zápis využitím vícevláknového přístupu.

Všechny vytyčené cíle této diplomové práce se podařilo realizovat a implementoval jsem úspěšně prototypovou aplikaci, jež obdobně jako projekt k-Wave provádí kroky simulace a neblokujícím způsobem ukládá data po každém kroku pomocí dedikovaných vláken. Následně získané poznatky byly využity také pro implementaci dvou verzí aplikace k-Wave využívající neblokujícího zápisu. První verze provádí zápis ze všech procesů a druhá nejprve kumuluje data pro zápis v rámci skupin a následný neblokující zápis je pak prováděn pouze jedním procesem z každé skupiny.

Obě implementované verze aplikace k-Wave byly testovány na různých simulačních scénářích a podařilo se za určitých podmínek dosáhnout zrychlení celkové doby běhu simulace až o 33% a tím docílit také snížení finanční náročnosti simulace. Vzhledem k přidané režii správy vláken a kumulaci dat však využití neblokujícího zápisu není univerzálním řešením pro libovolné simulační scénáře. V případě zápisu do souboru všemi MPI procesy je nutná podpora Intel Hyperthreading technologie na výpočetních uzlech. Pro neblokující zápis s kumulací dat platí, že v každém iteračním kroku musí být zapsáno alespoň 1.5MB dat do každého datasetu, aby se dosáhlo dostatečného zrychlení, jež vykompenzuje přidanou režii.

Díky této diplomové práci jsem se dozvěděl mnohé o architektuře superpočítačů a návrhu distribuovaných superpočítačových aplikací s důrazem na vysoký výkon a paralelní zpracování. Také jsem dostal unikátní příležitost se podílet na vývoji aplikace, jež využívá až desítky uzlů, stovky procesů a produkuje terabajty dat.

V budoucnu je možné tuto diplomovou práci rozšířit poté, když vyjde nová verze knihovny HDF5, jež umožní neblokující kolektivní vstupně výstupní operace, nebo alespoň nahradí blokující čekání při zápisových operacích neblokující variantou. Pokud budou implementovány neblokující vstupně výstupní operace v plném měřítku, bude možné odstranit explicitní vytváření vláken dedikovaných pro zápis. Avšak i pouhá náhrada blokujícího čekání při zápisových operacích by odstranila nutnost kumulování dat před zápisem a tím výrazně snížila režii.

V aplikaci k-Wave je také možné implementovat analýzu vstupní domény, subdomén a rozložení procesů na jednotlivých uzlech a následně zvolit vhodnou implementaci tříd pro výstupní proudy dat. Pro malé domény a subdomény je vhodnější zvolit blokující zápis,

v opačném případě neblokující zápis s kumulací dat. Analýza rozložení procesů na jednotlivých uzlech umožní implicitní zvolení velikosti skupin procesů pro kumulaci dat.

Další možností rozšíření této diplomové práce je náhrada kolektivních zápisových operací malých subdomén individuálními operacemi. V takovém případě by bylo možné paralelizovat více zápisů v jeden okamžik a výrazně by se snížila doba zápisu v každém iteračním kroku. Tato změna by opět vyžadovala implementaci analýzy rozložení dat na jednotlivých MPI procesech.

Literatura

[1] Argonne National Laboratory. *IOFSL*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.mcs.anl.gov/research/projects/iofsl/about/.

[2] Argonne National Laboratory. *A quick overview of MPI's send modes*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.mcs.anl.gov/research/projects/mpi/sendmode.html.

[3] Arm Holdings. *The Number One Debugger for C, C++ and Fortran Threaded and Parallel Code*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt.

[4] Arm Holdings. *Show Exactly Where and Why Code Is Losing Performance*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.arm.com/products/development-tools/server-and-hpc/forge/map.

[5] Beech Brandt, J. *When applications go exascale*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.epcc.ed.ac.uk/blog/2014/02/10/when-applications-go-exascale-cresta-project.

[6] Bradley Treeby, B. C. a Jaros, J. *A MATLAB toolbox for the time-domain simulation of acoustic wave fields*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: http://www.k-wave.org/.

[7] Cluster File Systems, Inc.. *Lustre: A Scalable, High-Performance File System*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf.

[8] Cluster File Systems, Inc.. *Lustre Object Storage Service*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: http://wiki.lustre.org/Lustre_Object_Storage_Service_(OSS).

[9] Editor Hosting Journalist. *ARM Extends HPC Offering with Acquisition of Software Tools Provider Allinea Software*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://hostingjournalist.com/cloud-hosting/arm-extends-hpc-offering-with-acquisition-of-software-tools-provider-allinea-software/.

[10] Elsevier B.V. . *Direct Memory Access*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.open-mpi.org/.

[11] Fox, A. *Why Aren't CPUs Getting Faster?* 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.applegazette.com/mac/why-arent-cpus-getting-faster/.

[12] Frigo, M. a Johnson, S. G. *FFTW*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: http://www.fftw.org/.

[13] German BMBF project SILC and US DOE project PRIMA. *Scalable Performance Measurement Infrastructure for Parallel Codes*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.vi-hps.org/projects/score-p/.

[14] Ghent University. *What is EasyBuild?* 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://easybuild.readthedocs.io/en/latest/Introduction.html.

[15] Gropp, W. *Introduction to MPI I/O*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf.

[16] Intel Corporation. *Intel Hyper-Threading Technology*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[17] IT4Innovations. *Available Salomon Modules - IT4Innovations Documentation*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://docs.it4i.cz/modules-salomon/.

[18] IT4Innovations. *Hardware Overview - IT4Innovations Documentation*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://docs.it4i.cz/salomon/hardware-overview/.

[19] IT4Innovations. *Resources Allocation Policy - IT4Innovations Documentation*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://docs.it4i.cz/general/resources-allocation-policy/.

[20] IT4Innovations. *Storage - IT4Innovations Documentation*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://docs.it4i.cz/salomon/storage/.

[21] IT4Innovations. *18th Open Access Grant Competition*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://www.it4i.cz/2019/10/18th-open-access-grant-competition/.

[22] Jaros, J., Treeby, E. B., Cox, B. T., Nandapalan, N., Johnston, B. and Rendell, A. P.. *Large-scale Ultrasound Simulations in Human Body*. 2014. Prezentace předmětu Architektura a programování paralelních systémů, Fakulta informačních technologií VUT Brno.

[23] Latham, R., Ross, R., Welch, B. a Antypas, K. *Parallel I/O in Practice*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.nersc.gov/assets/Training/pio-in-practice-sc12.pdf.

[24] OpenSFS and EOFS. *Introduction to Lustre Architecture*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf.

[25] Popo, G., Mastorakis, N. a Mladenov, V. *Calculation of the acceleration of parallel programs as a function of the number of threads*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.researchgate.net/publication/228569958.

[26] Progsch, J. a Zeman, V. *ThreadPool*. 2014. [Online; navštíveno 18.04.2020]. Dostupné z: https://github.com/progschj/ThreadPool/.

[27] Rosenberg, D. *Open-source Lustre gets supercomputing nod*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.cnet.com/news/open-source-lustre-gets-supercomputing-nod/.

[28] Rutman, N. *Rock-Hard Lustre Trends in Scalability and Quality*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: http://www.opensfs.org/wp-content/uploads/2011/11/Rock-Hard1.pdf.

[29] Schreiner, H. a Top, P. *CLI11: Command line parser for C++11*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://github.com/CLIUtils/CLI11.

[30] Terpstra, D., Jagode, H., You, H., Dongarra, J.. *Collecting Performance Data with PAPI-C*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: http://icl.cs.utk.edu/papi/index.html.

[31] The HDF Group. *HDF5 C/FORTRAN REFERENCE MANUAL*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://portal.hdfgroup.org/pages/viewpage.action?pageId=50073943.

[32] The HDF Group. *HDF5 Tool Interfaces*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://support.hdfgroup.org/HDF5/doc/RM/Tools.html.

[33] The HDF Group. *INTRODUCTION TO HDF5*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://portal.hdfgroup.org/display/HDF5/Introduction+to+HDF5#IntroductiontoHDF5-software.

[34] The HDF Group. *PARALLEL HDF5*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://support.hdfgroup.org/HDF5/PHDF5/.

[35] The HDF Group. *Questions about thread-safety and concurrent access*. 2020. [Online; navštíveno 18.04.2020]. Dostupné z: https://portal.hdfgroup.org/display/knowledge/Questions+about+thread-safety+and+concurrent+access.

[36] The Open MPI Project. *Open MPI: Open Source High Performance Computing*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.open-mpi.org/.

[37] University of Illinois at Urbana-Champaign. *Property Lists*. 2019. [Online; navštíveno 05.01.2020]. Dostupné z: https://www.asc.ohio-state.edu/wilkins.5/computing/HDF/hdf5tutorial/property.html.

Příloha A

Obsah DVD

```
/
├── Data/ ... Skripty v jazyce MATLAB pro generování
│             vstupnich souborů pro k-Wave
│
├── Jobs/ ... Skripty pro zadávání úloh na superpočítači
│             Salomon
│
└── k-Wave-batch/ ... Zdrojové soubory aplikace k-Wave
    │                 s přeposíláním dat a neblokujícím zápisem
    │                 z podmožiny procesů
    │                 Vypsané jsou pouze soubory editované v této
    │                 diplomové práci
    ├── Containers/ ... Správce dat
    │   │
    │   └── OutputStreamContainer.{cpp|h} ... Jednotné rozhraní
    │                                          pro veškeré zápisy
    │                                          subdomén do výstupního
    │                                          souboru
    │
    ├── KSpaceSolver/
    │   └── KSpaceFirstOrder3DSolver.{cpp|h} ... Hlavní běh simulace
    │
    ├── OutputStreams/ ... Selekce subdomén a zápis dat
    │   │                  do výstupního souboru
    │   │
    │   ├── IndexOutputStream.{cpp|h} ... Zápis subdomén
    │   │                                 reprezentovaných
    │   │                                 senzorovou maskou
    │   │
    │   └── CuboidOutputStream.{cpp|h} ... Zápis subdomén tvaru
    │                                       kuboidu
    │
    ├── Parameters/
    │   └── Parameters.{cpp|h} ... Správce parametrů
    │                              simulace
    │
    └── Utils/
```

```
│   └─ ThreadPool.h ... Fronta pro asynchronní
                         zápis pomocí vláken

    └─ Makefile ... Skript pro sestavení aplikace
                    k-Wave

├─ k-Wave-all/ ... Zdrojové soubory aplikace k-Wave
                   s neblokujícím zápisem ze všech procesů
                   Stejná struktura souborů jako k-Wave-batch

├─ Sources/ ... Zdrojové soubory prototypové aplikace
│   ├─ buffer.{cpp|h} ... Správce rotujících polí
│                         pro neblokující zápis
│   ├─ C11.hpp [29] ... Parser vstupních parametrů
│   ├─ common.h ... Pomocné struktury
│   ├─ input.{cpp|h} ... Generování vstupní domény
│   ├─ main.cpp ... Hlavní běh simulace
│   ├─ papi_cntr.h ... Vysokoúrovňová knihovna
│                      pro měření PAPI eventů
│   ├─ pattern.{cpp|h} ... Reprezentace subdomény
│                          pro zápis dat
│   └─ thread_pool.h [26] ... Fronta pro asynchronní zápis
│                             pomocí vláken

├─ Thesis/ ... Tato technická zpráva v LaTeX

├─ Misc/ ... Různé pomocné materiály
│   ├─ scorep-blocking.html ... Výsledek profilování
│                               prototypové aplikace
│                               (s blokujícím zápisem) pomocí
│                               Score-P
│   └─ scorep-nonblocking.html ... Výsledek profilování
                                    prototypové aplikace
                                    (s neblokujícím zápisem)
                                    pomocí Score-P

├─ Readme.md ... Průvodní soubor se základními informacemi
                 o projektu a návodem na sestavení a spuštění
                 implementovaných aplikací

└─ Makefile ... Skript pro sestavení prototypové aplikace
```