



VYHODNOCENÍ PARALELIZAČNÍCH PŘÍSTUPŮ PRO EFEKTIVNÍ NÁSOBENÍ ŘÍDKÝCH MATIC

Bc. Matěj Javorka

Diplomová práce
Fakulta informačních technologií
České vysoké učení technické v Praze
Katedra softwarového inženýrství
Studijní program: Informatika
Specializace: Softwarové inženýrství
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
6. května 2026



Zadání diplomové práce

Název:	Vyhodnocení paralelizačních přístupů pro efektivní násobení řídkých matic
Student:	Bc. Matěj Javorka
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2026/2027

Pokyny pro vypracování

- 1) Seznamte se s problematikou ukládání řídkých matic a analyzujte existující formáty pro jejich ukládání. [1–3]
- 2) Nastudujte a analyzujte algoritmy pro násobení řídkých matic jak v základních, tak pokročilých formátech jako ELL či BCRS. [1–7]
- 3) Pro alespoň 2 různé formáty implementujte v C++ sekvenční algoritmus pro násobení řídkých matic a také jeho paralelní verzi pomocí obou následujících různých paralelizačních přístupů:
 - a) C++ bez externích knihoven pro paralelizaci;
 - b) s použitím externí knihovny pro paralelizaci nad sdílenou pamětí.
- 4) Při návrhu dbejte na znovupoužitelnost, udržitelnost a snadnou rozšiřitelnost kódu. Prostudujte a diskutujte možnosti využití moderních vlastností jazyka ze standardu C++20. [8]
- 5) Všechna implementovaná řešení řádně otestujte a zautomatizujte kompilaci i ověření testů pomocí kontinuální integrace.
- 6) Změřte dobu běhu, pro paralelní řešení pak navíc i paralelní cenu, zrychlení a efektivnost. Porovnejte a diskutujte výkon jednotlivých implementovaných řešení.

Seznam odborné literatury:

- [1] SCOTT, Jennifer a TŮMA, Miroslav. Algorithms for Sparse Linear Systems. Springer Nature, 2023. ISBN 978-3-031-25820-6.
- [2] DUFF, Iain S.; ERISMAN, A. M. a REID, John Ker. Direct methods for sparse matrices / I.S.



Duff, A.M. Erisman, J.K. Reid. Numerical mathematics and scientific computation (Oxford). Oxford: Clarendon Press, c1986. ISBN 978-0198534211.

[3] OSTERBY, Ole a ZLATEV, Zahari. Direct Methods for Sparse Matrices. Lecture notes in computer science. Berlin: Springer-Verlag, 1983. ISBN 978-3-540-12676-8.

[4] JERZY S. RESPONDEK. Fast Matrix Multiplication with Applications. Springer Nature, 2025. ISBN 978-3-031-76930-6.

[5] YUSTER, Raphael a Uri ZWICK. Fast sparse matrix multiplication. ACM transactions on algorithms. 2005, roč. 1, č. 1, s. 2–13.

[6] HONG, Chuhe; WANG, Qinglin; MAO, Runzhang; LIANG, Yuechao; XIA, Rui et al. SaSpGEMM: Sorting-Avoiding Sparse General Matrix-Matrix Multiplication on Multi-Core Processors. Online. In: Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24). 2024, s. 1166-1175. Dostupné z: <https://doi.org/10.1145/3673038.3673054>.

[7] WOLFSON-POU, Jordi; LAUKEMANN, Jan a PETRINI, Fabrizio. MAGNUS: Generating Data Locality to Accelerate Sparse Matrix-Matrix Multiplication on CPUs. Online. In: Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25). 2025, s. 442-457. Dostupné z: <https://doi.org/10.1145/3721145.3725773>.

[8] NICOLAI M. JOSUTTIS. C++ 20: The Complete Guide. 2022. ISBN 978-3-96730-020-8.

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2026 Bc. Matěj Javorka. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Javorka Matěj. *Vyhodnocení paralelizačních přístupů pro efektivní násobení řídkých matic*. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2026.

Rád bych poděkoval doc. Ing. Ivanovi Šimečkovi, Ph.D., za jeho čas, pomoc, vstřícnost a odborné vedení této diplomové práce. Dále bych chtěl poděkovat středisku ELSA ČVUT za zastoupení studentů, kteří v životě neměli mnoho štěstí. Na závěr bych rád poděkoval své rodině za její soustavnou pomoc a oporu, díky které jsem se mohl přese všechny překážky a nepřízně osudu vydat po nevyšlapané cestě.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

Prohlašuji, že jsem v průběhu příprav a psaní závěrečné práce použil nástroje umělé inteligence. Vygenerovaný obsah jsem ověřil. Stvrzuji, že jsem si vědom, že za obsah závěrečné práce plně zodpovídám.

V Praze dne 6. května 2026

Abstrakt

Náplní této diplomové práce je analýza, návrh, implementace a vyhodnocení paralelních algoritmů pro násobení řídkých matic na architekturách se sdílenou pamětí za použití programovacího jazyka C++ a knihovny OpenMP.

V literární rešerši jsou nejprve popsány základní principy výpočetní architektury se sdílenou pamětí a také jsou představeny základní pojmy z teorie složitosti a lineární algebry. Následuje definování řídké matice, popis možných formátů pro reprezentaci řídkých matic v paměti a také popis souborových formátů pro jejich přenos a ukládání. Závěr literární rešerše se věnuje představení a analýze moderních algoritmů pro násobení řídkých matic.

Praktická část práce se zabývá návrhem a implementací algoritmu pro násobení řídkých matic. Implementována je sekvenční varianta, paralelní varianta v jazyce C++ za použití moderních funkcí ze standardu C++20 a druhá paralelní varianta s využitím externí knihovny OpenMP. Korektnost těchto implementací je ověřena pomocí testů, které jsou společně s kompilací automatizovány pomocí virtualizační technologie Docker a nástrojů pro kontinuální integraci. Závěr praktické části se věnuje měření vhodných metrik implementovaných řešení, porovnání výkonu a posouzení výhod a omezení každého paralelizačního přístupu. Detailní výsledky všech měření jsou uvedeny v příloze této práce.

Klíčová slova řídké matice, formáty řídkých matic, násobení řídkých matic, SpGEMM, paralelismus nad sdílenou pamětí, C++, OpenMP, měření výkonu paralelních aplikací

Abstract

This thesis focuses on the analysis, design, implementation, and evaluation of parallel algorithms for multiplying sparse matrices on shared-memory architectures using the C++ programming language and the OpenMP library.

The literature research first describes the basic principles of shared memory computing architecture and introduces basic concepts from complexity theory, and linear algebra. This is followed by definitions of sparse matrices, a description of possible formats for representing sparse matrices in memory, and a description of file formats for their transfer and storage. The conclusion of the literature research is devoted to the presentation and analysis of modern algorithms for multiplying sparse matrices.

The practical part of this thesis deals with the design and implementation of an algorithm for multiplying sparse matrices. A sequential version is implemented, as well as a parallel version in C++ using modern features from the C++20 standard, and a second parallel version utilizing the OpenMP external library. The correctness of these implementations is verified using tests that, together with the compilation, are automated using Docker virtualization technology and continuous integration tools. The conclusion of the practical part focuses on measuring appropriate metrics of the implemented solutions, comparing performance, and assessing the advantages and limitations of each parallelization approach. Detailed results of all measurements are provided in the appendix of this thesis.

Keywords sparse matrices, sparse matrix formats, sparse matrix-matrix multiplication, SpGEMM, shared-memory parallelism, C++, OpenMP, performance evaluation of parallel applications

Obsah

1	Úvod	1
1.1	Cíle práce	2
1.2	Struktura práce	3
2	Výpočetní architektury se sdílenou pamětí	5
2.1	Základní komponenty počítače	5
2.2	Paměťová hierarchie	6
2.3	Virtuální paměťový prostor	8
2.4	Operační systém	9
2.5	Vícejádrové procesory a multiprocesorové systémy	11
2.6	Paměťová koherence a falešné sdílení	14
3	Prostředí pro paralelní programování	16
3.1	Paralelní programování v C++	17
3.1.1	Paměťový model	17
3.1.2	Správa vláken	20
3.1.3	Vzájemné vyloučení	21
3.1.4	Atomické typy	22
3.1.5	Podmíněné proměnné	23
3.1.6	Paralelní algoritmy v STL	23
3.1.7	Pokročilé synchronizační prostředky	25
3.1.8	Asynchronní programování	26
3.2	OpenMP	27
3.2.1	Paměťový model	28
3.2.2	Programovací model	29
3.2.3	Paralelní oblasti	30
3.2.4	Datový paralelismus	31
3.2.5	Úlohový paralelismus	32
3.2.6	Synchronizační prostředky	33
4	Základní pojmy	36
4.1	Teorie složitosti	36
4.2	Lineární algebra	39
4.3	Řídké matice	42

5	Formulace násobení matic	44
5.1	Row-by-Column násobení matic	45
5.2	Column-by-Row násobení matic	47
5.3	Row-by-Row násobení matic	49
5.4	Column-by-Column násobení matic	51
5.5	Násobení řídkých matic	52
6	Formáty pro reprezentaci a ukládání řídkých matic	53
6.1	Úložná schémata řídkých matic	53
6.1.1	Coordinate Format	54
6.1.2	Diagonal Format	55
6.1.3	ELLPACK	56
6.1.4	Compressed Sparse Row	57
6.1.5	Compressed Sparse Column	59
6.1.6	Block Compressed Sparse Row	60
6.1.7	Další úložné formáty	61
6.2	Souborové formáty řídkých matic	62
6.2.1	Harwell–Boeing	63
6.2.2	Rutherford–Boeing	63
6.2.3	Matrix Market	64
7	Algoritmy násobení řídkých matic	66
7.1	Predikce velikosti výstupní řídké matice	67
7.1.1	Metoda horního odhadu	67
7.1.2	Metoda přesné alokace	69
7.1.3	Pravděpodobností metoda	70
7.1.4	Progresivní metoda	71
7.2	Rozdělení výpočetní zátěže	72
7.2.1	Blokové rozdělení výpočetní zátěže	72
7.2.2	Grafové rozdělení výpočetní zátěže	75
7.3	Akumulace mezivýsledků	76
7.3.1	SPA	77
7.3.2	ESC	78
7.3.3	Akumulátor založený na slévání	79
7.3.4	Akumulátor s haldou	80
7.3.5	Akumulátor s hešováním	81
7.3.6	LLA	82
7.3.7	MAGNUS	84
8	Návrh	86
8.1	Algoritmus násobení řídkých matic	87
8.2	Hešovací tabulka	92
8.3	Zásobárna vláken	94

9 Implementace	97
9.1 Hešovací tabulka	98
9.2 Zásobárna vláken	101
9.3 Řídké matice a jejich načítání a zápis	104
9.4 Hierarchie tříd	107
9.5 Prefixový součet	109
9.6 Konverze řídkých matic	110
9.7 Falešné sdílení a jeho eliminace	114
9.8 Předávání výsledků hodnotou a jejich optimalizace	116
9.9 CLI aplikace a její rozhraní	117
10 Testování, automatizace sestavení a kontinuální integrace	120
10.1 Testování	120
10.2 Automatizace sestavení	124
10.3 Kontinuální integrace	125
11 Měření a vyhodnocení	127
11.1 Specifikace cílového systému	127
11.2 Způsob měření	128
11.3 Výsledky a vyhodnocení	131
12 Závěr	142
12.1 Možnosti rozšíření	143
A Uživatelská příručka	145
A.1 Závislosti	145
A.1.1 Povinné závislosti	145
A.1.2 Volitelné závislosti	146
A.1.3 Použité externí knihovny	146
A.2 Sestavení	147
A.3 Použití	147
A.3.1 <code>seq_spgemm</code>	147
A.3.2 <code>omp_spgemm</code>	148
A.3.3 <code>par_spgemm</code>	149
A.3.4 <code>measurements</code>	149
A.3.5 Automatizace měření	150
A.3.6 Testy	151
B Data pro měření	152
B.1 Velmi malé matice	152
B.2 Malé matice	153
B.3 Střední matice	153
B.4 Velké matice	154
B.5 Velmi velké matice	154

C	Výsledky měření	155
C.1	Výsledky pro uzly s procesorem Fujitsu A64FX	156
C.1.1	Velmi malé matice	156
C.1.2	Malé matice	158
C.1.3	Střední matice	160
C.1.4	Velké matice	162
C.1.5	Velmi velké matice	164
C.1.6	Průměrné zrychlení	165
C.1.7	Časy výpočtu a zrychlení v závislosti na počtu vláken .	165
C.2	Výsledky pro uzel s procesorem AMD EPYC 7543P	166
C.2.1	Velmi malé matice	166
C.2.2	Malé matice	168
C.2.3	Střední matice	170
C.2.4	Velké matice	172
C.2.5	Velmi velké matice	174
C.2.6	Průměrné zrychlení	175
C.2.7	Časy výpočtu a zrychlení v závislosti na počtu vláken .	175
D	Další dokumenty	176
	Bibliografie	181
	Obsah přiloženého archivu	189

Seznam obrázků

2.1	Příklad tříúrovňové paměťové hierarchie uvnitř procesoru. Ilustrace inspirována mikroarchitekturou Intel Nehalem. [2]	8
2.2	Ukázka virtuálního paměťového prostoru procesu se třemi vlákny. Inspirováno ilustracemi z [2].	12
2.3	Ukázka problému paměťové koherence. Překresleno z [2].	14
3.1	Úloha jako datový kanál mezi dvěma komunikačními body. Překresleno z [11].	27
3.2	Ukázka fork-join modelu. Inspirováno ilustrací z [16].	29
4.1	Řídká matice Chen/pkustk13 pocházející z kolekce SuiteSparse Matrix Collection. [24] Vizualizace vygenerována pomocí nástroje <i>marc</i> . [25]	42
5.1	Vizualizace RbC násobení. Překresleno z [34].	45
5.2	Vizualizace CbR násobení. Překresleno z [34].	47
5.3	Vizualizace RbR násobení. Překresleno z [34].	49
5.4	Vizualizace CbC násobení. Inspirováno ilustracemi z [34].	51
6.1	Příklad reprezentace řídké matice ve formátu COO.	54
6.2	Příklad reprezentace řídké matice ve formátu DIA.	55
6.3	Příklad reprezentace řídké matice ve formátu ELL.	57
6.4	Příklad reprezentace řídké matice ve formátu CSR.	58
6.5	Příklad reprezentace řídké matice ve formátu CSC.	60
6.6	Příklad reprezentace řídké matice ve formátu BCSR.	60
6.7	Příklad obecné čtvercové řídké matice řádu 5 s 8 nenulovými reálnými prvky. [44]	64
7.1	Obecný postup výpočtu SpGEMM algoritmů. Překresleno z [1].	66
7.2	Ukázka použití metody horního a přesného odhadu počtu nenulových prvků matice $\mathbf{C}$. Překresleno z [1].	68
7.3	Ukázka reprezentace násobení matic prostřednictvím výpočetní krychle. Inspirováno ilustrací ze článku [1].	72
7.4	Tři varianty jednorozměrného rozdělení výpočetní krychle. Překresleno z [1].	73
7.5	Tři varianty dvourozměrného rozdělení výpočetní krychle. Překresleno z [1].	74

7.6	Ukázka trojrozměrného rozdělení výpočetní krychle. Překresleno ze článku [1].	74
7.7	Ukázka reprezentace SpGEMM násobení matic pomocí hypergrafu a bipartitního grafu. Překresleno z [1].	76
7.8	Ukázka použití SPA akumulátoru. Překresleno z [1].	77
7.9	Ukázka použití ESC akumulátoru. Překresleno z [45].	78
7.10	Ukázka akumulátoru se sléváním. Překresleno z [48].	79
7.11	Ukázka akumulátoru s haldou. Inspirováno ilustrací z [54].	81
7.12	Ukázka akumulátoru s hešováním. Inspirováno ilustrací z [1].	82
7.13	Ukázka použití LLA akumulátoru. Překresleno z [45].	83
7.14	Ukázka výpočtu algoritmu MAGNUS. Překresleno z [5].	84
8.1	Ukázka výpočtu pole <i>offsets</i> podle [60] a podle řešení navrhaného v kapitole 8.1.	89
8.2	Ukázka hešovací tabulky s otevřeným adresováním a lineárním přidáváním. Překresleno z [60].	91
8.3	Návrhový model zásobárny vláken.	96
9.1	UML diagram klíčových částí hešovací tabulky.	99
9.2	UML diagram zásobárny vláken.	101
9.3	UML diagram struktur pro COO a CSR formáty řídkých matic.	104
9.4	UML diagram původní třídní hierarchie na příkladu tříd pro konverzi matic.	108
9.5	UML diagram tříd pro konverzi matic po přepracování na šablonové metody.	109
9.6	Ukázka třífázového paralelního prefixového součtu. Inspirováno ilustrací z [70].	111
11.1	Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici GHS_psddef/hood z kategorie středních matic.	132
11.2	Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici Williams/webbase-1M z kategorie velkých matic.	133
11.3	Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici DIMACS10/delaunay_n24 z kategorie velmi velkých matic.	134
11.4	Srovnání časů výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 s knihovnou Eigen pro matici Williams/webbase-1M z kategorie velkých matic.	135
11.5	Průměrné zrychlení paralelních variant SpGEMM násobení pro jednotlivé skupiny řídkých matic na výpočetním uzlu s procesorem Fujitsu A64FX. Označení na horizontální ose odpovídají zkratkám skupin řídkých matic.	138

11.6	Průměrné zrychlení paralelních variant SpGEMM násobení pro jednotlivé skupiny řídkých matic na výpočetním uzlu s procesorem AMD EPYC 7543P. Označení na horizontální ose odpovídají zkratkám skupin řídkých matic.	138
11.7	Zrychlení paralelních variant SpGEMM násobení v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu s procesorem Fujitsu A64FX.	139
11.8	Zrychlení paralelních variant SpGEMM násobení v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu s procesorem AMD EPYC 7543P.	139

Seznam tabulek

11.1	Rozdělení matic pro měření podle řádu čtvercové matice. . . .	129
B.1	Matice vybrané pro měření z kategorie velmi malých matic. . .	152
B.2	Matice vybrané pro měření z kategorie malých matic.	153
B.3	Matice vybrané pro měření z kategorie středních matic.	153
B.4	Matice vybrané pro měření z kategorie velkých matic.	154
B.5	Matice vybrané pro měření z kategorie velmi velkých matic. . .	154
C.1	Časy výpočtu pro velmi malé matice na AArch64 uzlu.	156
C.2	Výkonnostní metriky OpenMP varianty pro velmi malé matice na AArch64 uzlu.	156
C.3	Výkonnostní metriky varianty s C++ vlákny pro velmi malé matice na AArch64 uzlu.	157
C.4	Časy výpočtu pro malé matice na AArch64 uzlu.	158
C.5	Výkonnostní metriky OpenMP varianty pro malé matice na AArch64 uzlu.	158
C.6	Výkonnostní metriky varianty s C++ vlákny pro malé matice na AArch64 uzlu.	159
C.7	Časy výpočtu pro střední matice na AArch64 uzlu.	160
C.8	Výkonnostní metriky OpenMP varianty pro střední matice na AArch64 uzlu.	160
C.9	Výkonnostní metriky varianty s C++ vlákny pro střední matice na AArch64 uzlu.	161
C.10	Časy výpočtu pro velké matice na AArch64 uzlu.	162
C.11	Výkonnostní metriky OpenMP varianty pro velké matice na AArch64 uzlu.	162

C.12 Výkonnostní metriky varianty s C++ vlákny pro velké matice na AArch64 uzlu.	163
C.13 Časy výpočtu pro velmi velké matice na AArch64 uzlu.	164
C.14 Výkonnostní metriky OpenMP varianty pro velmi velké matice na AArch64 uzlu.	164
C.15 Výkonnostní metriky varianty s C++ vlákny pro velmi velké matice na AArch64 uzlu.	164
C.16 Průměrné zrychlení v rámci jednotlivých skupin matic na uzlu AArch64.	165
C.17 Časy výpočtu paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu AArch64.	165
C.18 Zrychlení paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu AArch64.	165
C.19 Časy výpočtu pro velmi malé matice na x86_64 uzlu.	166
C.20 Výkonnostní metriky OpenMP varianty pro velmi malé matice na x86_64 uzlu.	166
C.21 Výkonnostní metriky varianty s C++ vlákny pro velmi malé matice na x86_64 uzlu.	167
C.22 Časy výpočtu pro malé matice na x86_64 uzlu.	168
C.23 Výkonnostní metriky OpenMP varianty pro malé matice na x86_64 uzlu.	168
C.24 Výkonnostní metriky varianty s C++ vlákny pro malé matice na x86_64 uzlu.	169
C.25 Časy výpočtu pro střední matice na x86_64 uzlu.	170
C.26 Výkonnostní metriky OpenMP varianty pro střední matice na x86_64 uzlu.	170
C.27 Výkonnostní metriky varianty s C++ vlákny pro střední matice na x86_64 uzlu.	171
C.28 Časy výpočtu pro velké matice na x86_64 uzlu.	172
C.29 Výkonnostní metriky OpenMP varianty pro velké matice na x86_64 uzlu.	172
C.30 Výkonnostní metriky varianty s C++ vlákny pro velké matice na x86_64 uzlu.	173
C.31 Časy výpočtu pro velmi velké matice na x86_64 uzlu.	174
C.32 Výkonnostní metriky OpenMP varianty pro velmi velké matice na x86_64 uzlu.	174
C.33 Výkonnostní metriky varianty s C++ vlákny pro velmi velké matice na x86_64 uzlu.	174
C.34 Průměrné zrychlení v rámci jednotlivých skupin matic na uzlu x86_64.	175
C.35 Časy výpočtu paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu x86_64.	175
C.36 Zrychlení paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu x86_64.	175

Seznam výpisů kódu

3.1	Ukázka použití vláken, zámku a podmíněné proměnné. Zdrojový kód převzat z [11].	24
3.2	Obecná syntaxe direktiv OpenMP v jazyce C++. [15]	30
3.3	Ukázka použití synchronizační direktivy <code>atomic</code>	34
6.1	Reprezentace matice z obrázku 6.7 ve formátu Matrix Market. [44]	65
9.1	C++ koncept <code>hashable_key</code> pro šablonové parametry hešovací tabulky s otevřeným adresováním.	100
9.2	Implementace metody <code>enqueue</code> pro vložení obecné úlohy do zásobárny vláken.	103
9.3	Implementace konceptů pro komplexní a aritmetické datové typy.	105
9.4	Implementace paralelního prefixového součtu s využitím konstrukce z OpenMP 5.0.	111
9.5	Ukázka kopírování nenulových prvků a jejich sloupcových indexů z COO matice do CSR formátu v lineárním čase.	113
9.6	Datová struktura zarovnaná na adresu dělitelnou velikostí bloku cache paměti.	115
9.7	Optimalizace NRVO na příkladu sekvenční konverze COO matice na CSR matici.	117
9.8	Ukázka aplikace odloženého vytvoření dočasného objektu.	117
10.1	Třída pro generování argumentů způsobem, jakým je operační systém předává programu.	122
10.2	Ukázka použití třídy <code>argv_builder</code>	123

Seznam zkratek

API	Application Programming Interface
BCRS	Block Compressed Row Storage
BCSR	Block Compressed Sparse Row
CbC	Column-by-Column
CbR	Column-by-Row
CLI	Command-Line Interface
CMR	Cache Memory Requirements
COO	Coordinate Format
CRS	Compressed Row Storage
CSR	Compressed Sparse Row
CPU	Central Processing Unit
DIA	Diagonal Format
DS	Dynamic Sparse Format
DMS	Distributed Memory System
DR	Data Reuse
ELL	ELLPACK
ESC	Expand-Sort-Contract
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
LLA	Linked List-Based Accumulator
MAC	Multiply-Accumulate
MMU	Memory Management Unit
MPI	Message Passing Interface
NRVO	Named Return Value Optimization
NUMA	Non-Uniform Memory Access
OS	Operating System
RAII	Resource Acquisition is Initialization
RAM	Random-Access Memory
RbC	Row-by-Column
RbR	Row-by-Row
SMP	Symmetric Multiprocessors
SMS	Shared Memory System
SPA	Sparse Accumulator
SpGEMM	Sparse General Matrix-Matrix Multiplication
SpMV	Sparse Matrix-Vector Multiplication
SSD	Solid State Drive
STL	Standard Template Library
TPP	Template Parameter Pack
UMA	Uniform Memory Access
UML	Unified Modeling Language



Kapitola 1

Úvod

„Programmers who care about performance must become parallel programmers, for sequential programs mean slow programs.“

— Patterson & Hennessy, *Computer Organization and Design*

Paralelní programování patří mezi královské disciplíny softwarového inženýrství a nabízí účinný způsob, jak zrychlit výpočetně náročné aplikace. Vývoj efektivních paralelních programů je však mimořádně náročný obor, který je velmi náchylný k chybám. Problémy, jako jsou časově závislé chyby, rozložení výpočetní zátěže a korektní synchronizace výpočetních vláken, vyžadují důkladnou analýzu a pečlivý návrh výsledného programu. Řešení těchto komplikovaných problémů proto posouvá paralelní programování z inženýrské a vědecké oblasti k umění. Přesto paralelní programování již dávno není výhradní doménou superpočítačů nebo vědeckých laboratoří. Kvůli nadále rostoucím nárokům na výpočetní výkon a díky neustálému vývoji hardwaru hraje paralelismus stále významnější roli i v každodenních aplikacích. S pokračujícím rozvojem vícejádrových procesorů bude navíc význam a potřeba paralelismu i v běžném softwaru dále růst.

Jedním z problémů, který významně těží z paralelizace výpočtu, je násobení řídkých matic (z angl. *sparse general matrix-matrix multiplication*, zkr. SpGEMM). Tento problém z numerické lineární algebry patří mezi jedny z nejdůležitějších stavebních kamenů v oblasti vědeckých výpočtů. Jeho efektivní implementace plní zásadní roli v teoretické informatice, jako například ve vícezdrojovém prohledávání grafu do šířky, barvení grafu, detekci a počítání cyklů nebo v nalezení nejkratších cest mezi všemi vrcholy grafu. Klíčové postavení má však také v nesupervizovaném strojovém učení, jako je Markovova shluková analýza, v aproximaci řešení rozsáhlých soustav lineárních rovnic, jako je algebraická metoda více sítí ale také v přírodních vědách, jako jsou simulace molekulární dynamiky nebo porovnávání a skládání genomů ve výpočetní biologii. Výkon těchto aplikací často zásadně závisí na efektivní implemen-

taci násobení řídkých matic. To je však výpočetně mimořádně náročné kvůli často velmi vysokým rozměrům vstupních matic a jejich nepravidelné struktuře, která významně komplikuje predikci přístupů do paměti, vyrovnávání výpočetní zátěže a předpovídání rozložení nenulových prvků výstupní matice. Tyto vlastnosti proto značně ztěžují návrh algoritmů, které efektivně využívají možnosti moderních paralelních architektur. [1]

Tato práce detailně přibližuje problematiku paralelizace násobení řídkých matic na architekturách se sdílenou pamětí. Seznamuje čtenáře s formáty pro reprezentaci a ukládání řídkých matic a analyzuje moderní algoritmy pro řešení problému jejich násobení. V praktické části je poté popsán návrh a implementace paralelního algoritmu pro násobení řídkých matic v jazyce C++ a také algoritmu využívajícího externí knihovnu pro paralelizaci OpenMP. Dále jsou představeny problémy, na které lze při implementaci narazit, a je popsáno, jak tyto problémy řešit. Pro porovnání obou paralelizačních přístupů je na vhodných testovacích maticích změřen paralelní čas, cena, zrychlení a efektivnost. Závěr diplomové práce pak porovnává výkon implementovaných řešení a poskytuje vhled do kompromisů mezi nativní paralelizací v jazyce C++ a přístupem založeným na externí knihovně OpenMP. Snad tato práce nejen pomůže čtenáři k hlubšímu pochopení problému paralelizace násobení řídkých matic, ale i v něm zažehne zájem věnovat se paralelnímu programování nebo řešení problémů numerické lineární algebry i nadále.

1.1 Cíle práce

Cílem této diplomové práce je analýza, návrh, implementace a vyhodnocení paralelních algoritmů pro násobení řídkých matic v programovacím jazyce C++ a pomocí knihovny OpenMP. Cíle práce bude dosaženo řešením několika vzájemně souvisejících dílčích cílů.

Prvním dílčím cílem je představení výpočetních architektur se sdílenou pamětí, definice řídkých matic a formátů pro jejich ukládání a reprezentaci v paměti. Součástí tohoto cíle je také detailní analýza moderních paralelních algoritmů pro násobení řídkých matic.

Druhým dílčím cílem této práce je návrh a implementace paralelního algoritmu pro násobení řídkých matic pomocí dvou různých paralelizačních technik. První technika se opírá o programovací jazyk C++ bez využití externích knihoven pro paralelizaci. Druhá technika pak využívá knihovnu OpenMP pro paralelizaci nad sdílenou pamětí. Se druhým cílem je spojeno i důkladné testování implementovaných algoritmů.

Posledním dílčím cílem je změření paralelního času, ceny, zrychlení a efektivnosti, a porovnání výkonu implementovaných řešení na vhodných vstupních datech. Tento cíl zahrnuje také vyhodnocení obou paralelizačních technik a posouzení výhod a omezení každého přístupu.

1.2 Struktura práce

Text této práce je rozdělen do 12 kapitol. Po úvodní kapitole následuje kapitola, která popisuje výpočetní architektury se sdílenou pamětí. Jsou popsány základní komponenty počítače, paměťová propast a hierarchie cache paměti uvnitř procesoru. Dále je popsán virtuální paměťový prostor procesu, jeho struktura a také operační systém a jeho role při vytváření a správě virtuálního paměťového prostoru procesu. Tato kapitola se dále zabývá popisem vícejádrových a multiprocessorových výpočetních systémů, jakož i popisem paměťové koherence a jejího vztahu k falešnému sdílení.

Třetí kapitola se věnuje popisu programovacích modelů nad sdílenou pamětí. Konkrétně popisuje prostředky pro vícevláknové programování ve standardní knihovně jazyka C++ včetně moderních funkcionalit z C++20, a také popisuje externí knihovnu pro programování nad sdílenou pamětí OpenMP.

Čtvrtá kapitola se zabývá základními pojmy z teorie složitosti a lineární algebry. Kromě asymptotické horní meze a amortizované složitosti jsou definovány také metriky pro porovnávání paralelních algoritmů, jako je paralelní čas, cena, zrychlení a efektivnost. Z lineární algebry jsou představeny pojmy vektor, jejich součet, skalární součin a lineární kombinace. Dále je definována matice, její vlastnosti a operace. Závěr této kapitoly se pak zabývá definicí pojmu řídká matice.

Pátá kapitola se zabývá detailní analýzou všech čtyř formulací násobení matic. U každé formulace jsou uvedeny její výhody a nevýhody, pseudokód, míra znovupoužití dat a nároky na cache paměť. Pro CbR, RbR a CbC formulace jsou také uvedeny formální důkazy jejich ekvivalence s RbC formulací.

Šestá kapitola představuje nejznámější formáty pro reprezentaci řídkých matic v paměti počítače, konkrétně se věnuje formátům COO, DIA, ELL, CSR, CSC a BCSR. Každý formát je detailně popsán, je uvedena jeho paměťová náročnost, výhody a nevýhody, a také ukázka reprezentace řídké matice v tomto formátu. Závěr kapitoly se věnuje popisu souborových formátů pro sdílení a ukládání řídkých matic Harwell–Boeing, Rutherford–Boeing a Matrix Market, u kterého je uvedena i ukázka reprezentace řídké matice v tomto formátu.

Sedmá kapitola se věnuje analýze postupu algoritmů pro násobení řídkých matic. Na začátku kapitoly je identifikováno pět různých fází obecného SpGEMM výpočtu a pro tři z těchto fází jsou detailně popsány a analyzovány dostupné metody, které problematiku dané fáze řeší, včetně jejich výhod a nevýhod. Pro fázi predikce velikosti výstupní řídké matice je popsána metoda horního odhadu, metoda přesné alokace, pravděpodobnostní metoda a progresivní metoda. Pro fázi rozdělení výpočetní zátěže je možné použít některou z blokových nebo grafových metod. Z blokových metod rozdělení výpočetní zátěže jsou představeny všechny metody jednorozměrného, dvourozměrného a třírozměrného způsobu rozdělení. Z grafových metod rozdělení jsou uvedeny metody reprezentace SpGEMM násobení prostřednictvím hypergrafu a bipar-

titního grafu. Pro fázi akumulace mezivýsledků je představen řídký akumulátor, ESC akumulátor, akumulátor založený na slévání, akumulátor s haldou, akumulátor s hešováním, LLA akumulátor a také způsob akumulace používaný v algoritmu MAGNUS.

Osmá kapitola se věnuje návrhu algoritmu pro paralelní násobení řídkých matic. Zvolený algoritmus je rozdělen na pět dílčích algoritmů, z nichž každý je detailně popsán, a jsou uvedeny rovněž pseudokódy těchto dílčích algoritmů. Tato kapitola se také věnuje návrhu hešovací tabulky pro akumulátor navrženého algoritmu a zásobárny vláken, která je použita pro paralelizaci SpGEMM algoritmu bez externích knihoven pro paralelizaci.

Devátá kapitola detailně popisuje implementaci všech tří variant aplikace pro násobení řídkých matic. V této kapitole je představena implementace hešovací tabulky, zásobárny vláken, struktur pro reprezentaci řídkých matic a také tříd pro jejich načítání a zápis do souboru. Dále se tato kapitola věnuje popisu hierarchie tříd celého projektu, implementaci paralelního prefixového součtu a také konvertorů pro převod řídkých matic. V této kapitole je také popsáno falešné sdílení a způsoby jeho eliminace v tomto projektu, a je také odůvodněna volba předávání vypočítaných výsledků hodnotou, včetně popisu optimalizací tohoto přístupu v moderním C++. Na závěr této kapitoly je popsáno CLI rozhraní implementovaných aplikací a implementace zpracování argumentů z příkazové řádky.

V desáté kapitole je popsán framework zvolený pro účely testování implementovaných tříd, včetně popisu maker, která jsou v rámci testování používána. Dále je popsán způsob automatizace sestavení aplikací, zpracování externích knihoven, vyhledání implementovaných testů a generování dokumentace prostřednictvím nástroje CMake. Závěr této kapitoly se věnuje kontinuální integraci těchto procesů v rámci fakultní instance webového repozitáře GitLab, včetně automatického nasazení vygenerované dokumentace na PagesFIT.

V jedenácté kapitole je popsána konfigurace výpočetních uzlů fakultního klastru ClusterFIT, které byly použity pro měření. Dále tato kapitola detailně popisuje způsob měření a rozdělení matic určených pro měření do pěti kategorií podle velikosti. Závěr této kapitoly se věnuje prezentaci a diskuzi naměřených výsledků.

Poslední dvanáctá kapitola se věnuje shrnutí celé diplomové práce a implementovaného projektu, včetně identifikace a popisu možností rozšíření této práce.

V příloze A jsou uvedeny uživatelské příručky všech tří variant aplikací pro SpGEMM násobení. Dále je uvedena příručka pro měřicí aplikaci i pro měřicí skript, který byl použit pro automatizaci měření na fakultním klastru. V příloze B je uveden seznam a popis matic, které byly vybrány pro měření, a v příloze C jsou pak uvedeny kompletní výsledky těchto měření.

Výpočetní architektury se sdílenou pamětí

Tato kapitola vysvětluje základní pojmy klíčové pro pochopení výpočetních architektur se sdílenou pamětí. Kapitola je koncipována tak, aby i čtenář bez hlubokých technických znalostí získal nezbytné informace, které mu pomohou pochopit problém paralelizace násobení řídkých matic nad sdílenou pamětí.

Na základní úrovni lze na počítač nahlížet jako na zařízení, které vykonává instrukce a zpracovává data podle předem definované posloupnosti pravidel, jež se nazývá program. Každý moderní počítačový systém zajišťuje komunikaci mezi čtyřmi základními komponentami, kterými jsou procesor (v angl. také *central processing unit*, zkráceně CPU), hlavní paměť, sekundární paměť a periferní zařízení.

2.1 Základní komponenty počítače

Procesor je zodpovědný za vykonávání instrukcí programu, tj. čte instrukce z paměti, interpretuje je a provádí odpovídající operace, jako jsou aritmetické výpočty, logické operace nebo přenos dat z hlavní paměti. Procesor se typicky skládá z aritmeticko-logické jednotky, která provádí aritmetické a logické operace, řadiče, který načítá a dekoduje instrukce z hlavní paměti, hodin, které synchronizují veškeré operace procesoru generováním periodického signálu, a registrů, které představují nejrychlejší paměť počítače. Data v registrech jsou procesoru k dispozici v každém cyklu CPU hodin, nicméně počet registrů je velmi omezený a s některými z nich může programátor pracovat přímo. Procesor však také obsahuje programátorovi skrytou, tzv. *cache* paměť, která bude popsána v následující kapitole. Procesory dále často obsahují samostatnou jednotku pro výpočty v plovoucí řádové čárce, jednotku pro vektorové operace nebo jednotku pro predikci skoků v programu, která zvyšuje

výkon procesoru tím, že umožňuje tzv. spekulativní provádění instrukcí. Moderní procesory pro zvýšení výkonu typicky také implementují paralelismus na úrovni instrukcí, neboť využívají jak paralelismus v čase (tzv. *pipelining*), tak paralelismus v prostoru (superskalární zřetězení). [2, 3, 4]

Hlavní paměť (známá také pod označením RAM, z angl. *random access memory*) je úložiště, ve kterém jsou uloženy instrukce programu a kam se během jeho běhu ukládají data programu. Paměť je organizována jako posloupnost adresovatelných jednotek o velikosti 1 bajt, tj. 8 bitů a každý bajt má svou unikátní fyzickou adresu. Moderní procesory používají 64-bitovou architekturu a jsou tedy teoreticky schopné adresovat až 2^{64} bajtů, ačkoliv reálné implementace této architektury, jako je x86_64, typicky používají fyzické paměťové adresy kratší délky, např. 48 bitů. Hlavní paměť je volatilní, což znamená, že její obsah je při vypnutí počítače ztracen. Naproti tomu sekundární paměť, jako jsou HDD a SSD disky, je nevolatilní a obvykle uchovává operační systém, programy a uživatelská data. Sekundární paměť je však výrazně pomalejší oproti hlavní paměti. Zatímco přístupový čas hlavní paměti se pohybuje v desítkách nanosekund, v případě SSD disků se jedná o desítky mikrosekund a u HDD disků dokonce o jednotky až desítky milisekund. Z těchto důvodů musí být instrukce programu a data vždy nejprve načtena ze sekundární paměti do hlavní paměti počítače, odkud jsou pak přes adresní a datovou sběrnici načítána procesorem. [2, 3, 4]

Grafická a síťová karta pak společně se vstupně-výstupními zařízeními patří mezi periferní zařízení.

2.2 Paměťová hierarchie

Rychlost hlavní paměti je výrazně nižší než rychlost CPU. Zatímco procesory se za posledních 45 let zrychlily téměř milionkrát, hlavní paměť se za stejné časové období zrychlila o něco více než desetkrát. Tento obrovský rozdíl je známý jako paměťová propast mezi výkonností CPU a RAM. Paměťová propast tvoří úzké hrdlo omezující výpočetní výkon počítačových systémů a patří mezi hlavní problémy počítačové architektury již několik desetiletí. Tato rostoucí propast pak způsobuje, že procesor často tráví více času čekáním na data než prováděním potřebných výpočtů. Aby se tento problém zmírnil, vkládá se mezi CPU a hlavní paměť několik úrovní rychlejší a menší paměti, která se označuje jako cache paměť. [2, 3, 4]

Cache paměť uchovává kopie dat z hlavní paměti, přičemž základní princip její efektivity spočívá v pozorování, že programy většinou nepřistupují k položkám v hlavní paměti náhodně, ale typicky v daném okamžiku přistupují jen k malé části instrukčního a datového adresového prostoru. Konkrétně programy vykazují vzorce časové a prostorové lokality, které může cache paměť efektivně využívat. Časová lokalita označuje princip, že položky, ke kterým se přistupovalo nedávno, budou brzy zapotřebí znovu. Podobně prostorová lokalita znamená, že položky, které se nacházejí v paměti poblíž právě po-

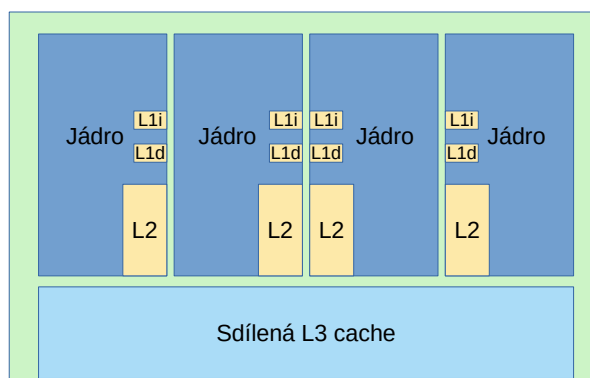
užívaných, budou brzy zapotřebí také. Paměťový systém proto do procesoru z hlavní paměti nenačítá pouze požadovanou položku, ale celý blok po sobě jdoucích bajtů, a tento blok uloží do cache paměti procesoru. Na hlavní paměť je proto možné nahlížet jako na oblast dat rozdělenou na číslované bloky cache paměti. Zatímco velikost hlavní paměti dosahuje desítek gigabajtů, celková velikost cache paměti se pohybuje v jednotkách nebo nižších desítkách megabajtů. Cache paměť je proto schopna v daném okamžiku uchovávat velmi omezený počet bloků hlavní paměti. Pokud je cache paměť plná a je zapotřebí načíst z hlavní paměti nový blok dat, musí se procesor pomocí vhodné strategie pro náhradu bloků rozhodnout, který ze stávajících bloků cache paměti vyřadit. [2, 3, 4]

Čím je paměť rychlejší, tím větší plochu zabírá. Pro procesory to tedy znamená, že rychlejší paměť musí mít menší kapacitu. Procesory tento problém řeší pomocí hierarchie cache pamětí, přičemž každá úroveň je větší a pomalejší než předchozí. Dnešní procesory většinou mívají následující tříúrovňovou hierarchii. [2, 3, 4]

- *L1 cache* je nejrychlejší a nejmenší cache paměť. Její velikost obvykle dosahuje pár desítek kilobajtů a na rozdíl od vyšších úrovní bývá rozdělena na instrukční (L1i) a datovou (L1d) část. Ve vícejádrových procesorech je L1 cache paměť replikována pro každé jádro zvlášť. [2, 3]
- *L2 cache* dosahuje velikosti několika stovek kilobajtů a ve vícejádrových procesorech má každé jádro většinou opět svou vlastní instanci. [2, 3]
- *L3 cache* má velikost několika megabajtů a tato úroveň je společná pro více jader procesoru. [2, 3]

Možná implementace nejnižších tří úrovní paměťové hierarchie uvnitř procesoru se čtyřmi jádry inspirované mikroarchitekturou Intel Nehalem je k vidění na obrázku 2.1.

Když procesor požaduje data z určité adresy hlavní paměti, nejprve se ověří, zda se požadovaná data nacházejí v cache paměti procesoru. Pokud jsou požadovaná data již načtena v cache paměti, jsou vrácena velmi rychle. Například L1 cache paměť je pouze o pár jednotek hodinových cyklů pomalejší než registry procesoru. Pokud požadovaná data v cache paměti nejsou, je potřeba je načíst z hlavní paměti, což je z hlediska času drahá operace, protože latence hlavní paměti bývá v řádu vyšších desítek hodinových cyklů. Je tedy zřejmé, že ačkoli je cache paměť pro programátora transparentní, její efektivní využívání pomocí principů časové a prostorové lokality je naprosto klíčové pro výkon moderních algoritmů, a to obzvlášť u výpočetně intenzivních problémů, jako je násobení řídkých matic. Například hlavní myšlenka algoritmu MAGNUS pro paralelní násobení řídkých matic spočívá v důmyslném a komplexním využívání cache pamětí. [2, 5]



■ **Obrázek 2.1** Příklad tříúrovňové paměťové hierarchie uvnitř procesoru. Ilustrace inspirována mikroarchitekturou Intel Nehalem. [2]

2.3 Virtuální paměťový prostor

V předchozí části bylo popsáno, jak programy přistupují k hlavní paměti za účelem načítání dat a instrukcí. Skutečný mechanismus, jakým programy interagují s pamětí, je však mnohem složitější. Programy totiž nepřistupují přímo k fyzickým adresám hlavní paměti, ale pracují s adresami virtuálního paměťového prostoru. Virtuální adresy jsou pak za podpory hardwaru překládány na skutečné fyzické adresy. [2]

Bez virtuálního adresového prostoru by nebylo možné, aby na jednom systému běželo více programů souběžně. Programy by totiž v takovém případě musely vědět, kde se v paměti nacházejí ostatní programy, aby nedošlo k nechtěnému nebo záměrnému zásahu do dat jiného programu nebo dokonce samotného operačního systému. Virtuální paměť tyto problémy řeší tím, že každému programu přiděluje jeho vlastní izolovaný virtuální paměťový prostor. Teoretická velikost virtuálního adresového prostoru je významně vyšší než velikost hlavní paměti. Na 64-bitových systémech je tato velikost až 2^{64} bajtů, ačkoli reálné implementace většinou používají virtuální adresy kratší délky, např. 48 bitů. Každý program má tedy iluzi, že má k dispozici velký souvislý paměťový prostor, a když je tento program spuštěn, překládá komponenta procesoru MMU (*memory management unit*) každou virtuální adresu na odpovídající fyzickou adresu v hlavní paměti. Díky tomuto mechanismu oddělených virtuálních paměťových prostorů může jednotka MMU v součinnosti s operačním systémem snadno zabránit tomu, aby jeden program přistupoval k fyzické paměti jiného programu. [2, 6, 7, 8]

Podobně jako je z hlediska cache paměti hlavní paměť rozdělena na bloky, je i v tomto případě virtuální a tím i hlavní paměť rozdělena na bloky, které se označují jako stránky. Virtuální adresový prostor je rozdělený na virtuální stránky, zatímco fyzická paměť je rozdělena na fyzické stránky, které se také nazývají rámce. Tento mechanismus stránkování umožňuje přesouvání stránek programů v hlavní paměti nebo jejich odkládání do sekundární paměti v pří-

padě, že se hlavní paměť zaplní. Virtuální paměť tedy umožňuje programům používat více paměti, než je fyzicky k dispozici, tím, že méně často používané stránky jsou uchovávány v sekundární paměti. MMU pak udržuje mapování mezi virtuálními a fyzickými stránkami pomocí tabulky stránek. Pokud chce program přistoupit k virtuální adrese, MMU pomocí tabulky stránek určí, který fyzický rámec v hlavní paměti odpovídá dané virtuální stránce, případně načte ve spolupráci s operačním systémem odloženou stránku ze sekundární do hlavní paměti, a poté provede překlad na fyzickou adresu. Pokud program přistupuje na virtuální adresu, která nespadá do programem alokované části jeho virtuálního adresového prostoru, tj. adrese neodpovídá žádný fyzický rámec v hlavní paměti ani stránka odložená v sekundární paměti, je běh tohoto programu operačním systémem předčasně ukončen. [2, 6, 7]

Pro účely této práce je důležité, že virtuální paměťový prostor je sdílený mezi všemi výpočetními vlákny daného procesu.

2.4 Operační systém

Operační systém (dále také OS) je softwarová vrstva, která spravuje hardwarové prostředky a poskytuje služby spuštěným programům, tj. je prostředníkem mezi programy a hardwarem počítačového systému. Operační systém tedy mimo jiné rozhoduje, kterým programům budou v daném okamžiku přidělena výpočetní jádra CPU, a ve spolupráci s MMU zajišťuje správu virtuální paměti právě spuštěných programů. [6, 7, 8]

Zatímco program představuje spustitelný binární soubor, který je uložený v sekundární paměti, proces je konkrétní instancí spustitelného programu. Jedná se tedy o entitu, které jsou alokovány systémové prostředky, jako je virtuální paměťový prostor, vlákna, sada otevřených souborů a síťová spojení. Procesy jsou však kvůli alokaci systémových prostředků poměrně nákladné abstrakce. Naproti tomu vlákno je sekvencí výpočtu v rámci jednoho konkrétního procesu, přičemž jádro procesoru je přidělováno konkrétnímu vláknu a nikoli procesu. Každý proces má implicitně jedno výpočetní hlavní vlákno, označované také jako *main* vlákno, a další vlákna je pak možné vytvářet a spravovat prostřednictvím systémových volání, standardních knihoven programovacích jazyků (jako je třída *std::thread* jazyka C++) nebo externích knihoven typu OpenMP. Vlákna vytvořená v rámci procesu sdílí většinu prostředků, které byly alokovány tomuto procesu, jako je virtuální paměťový prostor, přístup k souborům a síťová spojení. [6, 7, 8]

Vlákna a operační systém sdílí omezený počet procesorových jader výpočetního systému, přičemž jedno vlákno může být v daném okamžiku zpracováváno jedním jádrem procesoru. Z těchto důvodů je nezbytné, aby se vlákna na výpočetních jádrech střídala. Mechanismus, při kterém se vlákna střídají v používání jader CPU, se nazývá přepínání kontextu. Kontext představuje všechny potřebné informace, které jsou nezbytné pro opětovné spuštění přeru-

šeného vlákna od okamžiku jeho přerušení, jako jsou obsahy registrů procesoru, čítač instrukcí, který označuje, jakou instrukci dané vlákno aktuálně provádí, a ukazatel na zásobník, který bude popsán později. [6, 7, 8]

Díky sdílení prostředků mezi vlákny daného procesu vyžaduje vytvoření nového vlákna výrazně nižší režii v porovnání s procesem. Také přepínání mezi vlákny v rámci stejného procesu je rychlejší než přepínání mezi procesy, protože není nutné měnit tabulku stránek. Samotná vlákna žijí v iluzi, že jsou spuštěna bez přerušení od začátku výpočtu až do konce. V operačním systému je většinou implementováno několik strategií pro přepínání vláken a výběr vhodného vlákna, kterému bude umožněno po určitou dobu využívat jádro procesoru poté, co se jádro CPU uvolní. Procesorové jádro se uvolní v případě, že aktuální vlákno dokončilo svůj výpočet, čeká na vstupně-výstupní operaci nebo již na jádře procesoru vyčerpalo svůj čas, který byl vláknu přidělen operačním systémem. Přepínání kontextu probíhá tak, že operační systém uloží kontext přerušeného vlákna, naplánuje další vlákno, kterému má být přidělen čas na jádře procesoru, nastaví kontext tohoto vlákna a obnoví jeho chod na jádře procesoru. Prostřednictvím přepínání kontextu tak operační systém umožňuje zdánlivě souběžný chod více programů i na jednojádrovém procesoru, kdy každému vláknu přidělí malé časové kvantum, což vytváří iluzi, že všechna vlákna běží souběžně, ačkoliv jádro procesoru může v daném okamžiku využívat pouze jedno vlákno. Na vícejádrových a víceprocesorových systémech pak operační systém koordinuje přidělování všech výpočetních jader vláknům běžících procesů. [6, 7, 8]

Při vytvoření nového procesu je jeho virtuální paměťový prostor uspořádán do několika následujících odlišných oblastí:

- *Textový segment* obsahuje spustitelné instrukce programu, tj. zkompileovaný zdrojový kód.
- *Datový segment* uchovává inicializované globální a statické proměnné programu.
- *BSS segment* (z historických důvodů pojmenovaný z angl. *Block Started by Symbol*) obsahuje neinicializované globální a statické proměnné. Všechny proměnné v tomto regionu jsou po vytvoření procesu explicitně inicializovány na nulu. [6]
- *Halda* je oblast používaná pro dynamickou alokaci paměti. Halda roste směrem k vyšším adresám s tím, jak je alokováno více paměti, a zmenšuje se, když se alokovaná paměť uvolňuje.
- Oblast určená pro knihovny sdílené mezi programy.
- Každé vlákno má svůj vlastní *zásobník*, který roste směrem k nižším adresám. Na zásobník se ukládají aktivační záznamy (nebo také rámce), které

umožňují volací a návratový mechanismus volání funkcí a procedur. Aktivacíní záznam zahrnuje například lokální proměnné, parametry funkcí a procedur, jejich návratové hodnoty a také návratové adresy. Když vlákno zavolá funkci nebo proceduru, je na zásobník daného vlákna vložen nový rámec, a když se funkce či procedura vrátí, je rámec odstraněn. Protože každé vlákno má svůj vlastní zásobník, mohou vlákna udržovat své vlastní lokální proměnné a volat funkce nezávisle na sobě. Zásobníky různých vláken jsou umístěny v různých oblastech virtuálního adresového prostoru a jsou odděleny ochrannými stránkami (z angl. *guard pages*) pro detekci přetečení maximální velikosti zásobníku. [6]

- Horní část virtuálního adresového prostoru je rezervována pro jádro operačního systému, přičemž do této oblasti nemá kód programu přístup. Tuto oblast využívá operační systém pro práci se svými vlastními datovými strukturami, když dojde k systémovému volání. [6, 8]

Operační systém také hlídá, aby vlákna přistupovala pouze do oblastí virtuálního paměťového prostoru, které byly řádně alokovány. Pokud chce vlákno např. přistoupit na adresu, která přesahuje aktuální velikost zásobníku nebo haldy, pak takové virtuální adrese nebude odpovídat žádná fyzická adresa a celý proces bude z bezpečnostních důvodů předčasně ukončen, jak bylo vysvětleno v kapitole 2.3. Uspořádání virtuálního paměťového prostoru procesu se třemi vlákny ukazuje obrázek 2.2.

Paralelním programováním nad sdílenou pamětí se tedy rozumí paralelní výpočet, kde několik vláken prostřednictvím několika výpočetních jader jednoho nebo více procesorů společně pracuje na řešení konkrétního problému, přičemž vlákna spolu komunikují prostřednictvím sdíleného virtuálního adresového prostoru. Tato snadná komunikace prostřednictvím sdílené paměti však s sebou také přináší synchronizační problémy, časově závislé chyby, riziko uváznutí vláken nebo na vícejádrových a multiprocesorových systémech také falešné sdílení, které bude popsáno v další části. Z těchto důvodů paralelní programování vyžaduje pečlivý návrh výsledného algoritmu a důkladné používání synchronizačních prostředků, mezi které patří například zámky, atomické operace nebo bariéry. [6, 8]

Pro paralelní násobení řídkých matic jsou však vlákna výhodnější než procesy, protože vlákna mohou mezi sebou sdílet vstupní matice a přístup k dalším datovým strukturám bez režijních nákladů spojených s meziprocesovou komunikací.

2.5 Vícejádrové procesory a multiprocesorové systémy

Po několik desetiletí se výkon procesorů zvyšoval především zvyšováním frekvencí, tj. taktu hodin procesoru, což s sebou přinášelo rychlejší provádění in-



■ **Obrázek 2.2** Ukázka virtuálního paměťového prostoru procesu se třemi vlákny. Inspirováno ilustracemi z [2].

strukcí. Kolem roku 2005 však tento přístup narazil na zásadní fyzikální problém, který se v angličtině označuje jako *power wall*, tedy energetická propast. Ukázalo se, že energetická náročnost roste rychleji než frekvence procesoru, a vzhledem k tomu, že většina této energie se v procesoru přeměňuje na teplo, začaly se nároky na chlazení zvyšovat natolik, že se staly nepraktickými. [2, 3]

Místo zrychlování frekvence jader proto začali výrobci procesorů umísťovat více jader na čip procesoru. Vícejádrový procesor (v angl. *multicore processor*) tedy obsahuje 2, 4, 8 nebo více nezávislých jader, přičemž tato jádra jsou schopna vykonávat souběžně svůj vlastní proud instrukcí. Jak již bylo vysvětleno v kapitole 2.2, každé jádro většinou má svou privátní L1 a L2 cache paměť. L3 cache paměť je pak sdílená mezi jádry procesoru společně s přístupem k hlavní paměti, což umožňuje, aby vlákna stejného procesu běžící na různých jádrech spolu mohla komunikovat prostřednictvím sdílené paměti. Procesory, které jsou určeny pro vysokou úroveň paralelizace na výpočetních klastrech (viz. dále) a obsahují vysoký počet výpočetních jader, se někdy označují jako mnohójádrové (z angl. *manycore processors*). [2, 3]

Kromě navyšování výpočetních jader je možné zvýšit propustnost systému navýšením počtu samotných procesorů, přičemž takové systémy se pak nazývají multiprocessorové. Tyto systémy se dělí na systémy se sdílenou pamětí (z angl. *shared memory systems*, zkr. SMS) a na systémy s distribuovanou pamětí (z angl. *distributed memory systems*, zkr. DMS). Na multiprocessorovém

SMS systému běží jedna instance operačního systému, zatímco DMS systém se skládá ze samostatných výpočetních uzlů a každý uzel má svou vlastní instanci operačního systému. Protože na SMS systému běží jedna instance operačního systému, jsou jejich programovacím modelem stejné prostředky, jaké byly popsány již v kapitole 2.4, tj. systémové knihovny pro práci s vlákny, jako např. POSIX Threads, standardní knihovny programovacích jazyků, jako např. `std::thread` jazyka C++ nebo externí knihovny typu OpenMP. V DMS systémech je potřeba zajistit komunikaci mezi jednotlivými výpočetními uzly pomocí mechanismu přijímání a zasílání zpráv, přičemž standardem pro programování DMS systémů je MPI (z angl. *Message Passing Interface*). Oba přístupy se pak běžně kombinují, např. SMS systémy jsou používány jako výpočetní uzly, které jsou vzájemně propojeny rychlými sítěmi s nízkou latencí do distribuovaného systému, který se nazývá výpočetní klastr. U výpočetních klastrů se pak používá MPI pro paralelismus napříč uzly a OpenMP pro paralelismus uvnitř uzlu. [2, 3]

Multiprocesorové SMS systémy mohou mít sdílenou fyzickou paměť, tj. hlavní paměť je centralizovaná, nebo oddělenou fyzickou paměť, kdy je hlavní paměť fyzicky rozdělena do několika regionů (nebo také uzlů), přičemž každý uzel je přidružen ke specifické podmnožině procesorů. U SMS systémů s oddělenou fyzickou pamětí tedy systém pracuje s virtuálně sdílenou pamětí, kdy každý procesor má svoji lokální paměť, ale zároveň může každý procesor přistupovat i do jiných paměťových uzlů. Je zřejmé, že u SMS systémů s oddělenou fyzickou pamětí bude čas přístupu do paměti různý, proto se multiprocesorové systémy z tohoto hlediska dělí na UMA (z angl. *uniform memory access*) a NUMA (z angl. *non-uniform memory access*) systémy. V UMA systémech trvá přístup do paměti ze všech procesorů stejně dlouho, zatímco v NUMA systémech je doba přístupu do paměti různá pro různé adresy, tj. pro různá místa v paměti, protože záleží na tom, zda procesor přistupuje ke své lokální paměti, nebo k jinému paměťovému uzlu. Kvůli vysokým nárokům na propustnost paměťové sběrnice jsou však UMA systémy špatně škálovatelné, proto UMA systémy obvykle mívají jednotky nebo nižší desítky procesorů, a systémy s vyšším počtem procesorů používají architekturu NUMA. Nejběžnějším multiprocesorovým systémem je SMP (z angl. *symmetric multiprocessors*) systém, který označuje multiprocesorový systém používající identické procesory a jedinou centrální sdílenou hlavní paměť. SMP systémy jsou tedy typu UMA a často slouží jako uzly pro výpočetní klastry, které obsahují stovky až desítky tisíc výpočetních SMP systémů. [2, 3]

Část měření v rámci této práce bude prováděna na výpočetních ARM64 uzlech, které jsou součástí fakultního klastru ClusterFIT. Každý tento uzel používá procesor Fujitsu A64FX a jedná se o SMS výpočetní systém typu NUMA, který disponuje 32GB hlavní pamětí a 4 NUMA uzly uvnitř CPU s 12 výpočetními jádry na jeden paměťový uzel, tj. celkem 48 jádry. [9]

Krok	Událost	Obsah cache jádra A	Obsah cache jádra B	Obsah oblasti X v hlavní paměti
0				0
1	Jádro A čte X	0		0
2	Jádro B čte X	0	0	0
3	Jádro A ukládá 1 do X	1	0	1

■ **Obrázek 2.3** Ukázka problému paměťové koherence. Překresleno z [2].

2.6 Paměťová koherence a falešné sdílení

V SMS systémech tedy platí, že si výpočetní jádra drží pohled na hlavní paměť optikou své vlastní soukromé nebo lokální sdílené cache paměti, tj. více jader najednou může mít uloženou kopii téže paměťové oblasti v bloku své cache paměti. Pokud jedno z jader tuto oblast prostřednictvím své cache paměti modifikuje, budou mít ostatní jádra ve svých cache pamětech zastaralou hodnotu. Je proto třeba zajistit, aby všechna jádra měla jednotný pohled na sdílená data, tj. je potřeba zabránit situaci, kdy dvě různá jádra prostřednictvím bloků svých cache pamětí vidí dvě různé hodnoty téže proměnné. Ukázka této situace je k vidění na obrázku 2.3. [2]

Tato problematika je známá jako problém paměťové koherence. Formální definice koherence paměťového systému je poměrně komplexní a lze ji nalézt například v [2], nicméně neformálně lze paměťový systém označit jako koherentní, pokud je zaručeno, že každé čtení sdílené buňky vrátí hodnotu naposledy zapsanou do této buňky. K zajištění koherence se používají hardwarové koherenční protokoly, které sledují a zneplatňují celé bloky cache paměti. Koherenční protokoly zaručují, že pokud jedno jádro modifikuje paměťovou oblast prostřednictvím bloku své cache paměti, pak jsou cache bloky odpovídající této oblasti v hlavní paměti zneplatněny v ostatních jádrech, která jsou tímto krokem přinucena znovu načíst aktualizovaný blok z hlavní paměti. Mezi nejrozšířenější koherenční protokoly patří MESI, MESIF a MOESI. [2, 3]

Problém, který úzce souvisí s paměťovou koherencí a může mít významný negativní vliv na výkon paralelních algoritmů, je falešné sdílení. Falešné sdílení nastává, když různá vlákna zapisují sice na různé adresy paměťového prostoru, avšak tyto adresy jsou natolik blízké, že jsou mapovány do stejného bloku cache paměti, ačkoliv vlákna žádná data nesdílejí. Pokud vlákna přistupují k různým proměnným, které jsou však mapovány do stejného cache bloku, a jedno z vláken aktualizuje hodnotu své proměnné, automaticky tím prostřednictvím koherenčního protokolu zneplatní cache bloky ostatních jader. Pro paralelní algoritmy je to jeden z klíčových problémů, protože ačkoliv může být paralelní algoritmus navržený tak, aby výpočetní vlákna pracovala v disjunktních oblastech sdílené paměti, mohou vlákna vzájemným opakovaným zneplatňováním

svých cache bloků zcela eliminovat rychlostní efekt cache pamětí, způsobit částečnou serializaci paralelního výpočtu a výrazně tak degradovat výkon celého programu. Řešením tohoto problému je zajistit, aby vlákna pracovala v disjunktních cache blocích, čehož se docílí zarovnáním dat na velikost cache bloku a rozdělením výpočtu mezi vlákna po jednotlivých cache blocích. [2, 3]

■ ■

Prostředí pro paralelní programování

Tato kapitola se zabývá programováním vícevláknových aplikací, poskytuje přehled běžných prostředí pro paralelní programování a jejich vlastnosti. Detailněji je pak popsáno vícevláknovému programování v jazyce C++ a knihovně OpenMP.

Paralelní programování se zabývá vyjádřením výpočtů způsobem, který umožňuje souběžné provádění dílčích úkolů na více výpočetních jednotkách. Pro paralelní výpočet je klíčové, zda pro daný problém existuje využitelná možnost souběžného zpracování. Tedy je nutné, aby byl daný problém rozložitelný na dílčí podproblémy, které lze provádět souběžně. Toto rozložení na podproblémy však musí být využitelné, tj. musí existovat způsob, jak strukturovat zdrojový kód programu, aby bylo možné podproblémy namapovat na výpočetní jednotky a zajistit správnou koordinaci mezi nimi. [10]

Prostředí pro paralelní programování poskytuje základní nástroje a aplikační programovací rozhraní (z angl. *application programming interface*, zkr. API) potřebná k vytvoření paralelního programu. Tato práce se zabývá paralelním programováním nad sdílenou pamětí, tj. vícevláknovým programováním. Existuje poměrně mnoho prostředí pro paralelní programování systémů se sdílenou pamětí, tabulku s kompletním přehledem těchto prostředí z poloviny devadesátých let lze nalézt např. v [10]. Pro účely tohoto textu jsou důležitá tři prostředí paralelního programování.

POSIX Threads API, nebo také Pthreads, je nízkourovňové prostředí paralelního programování v jazyce C. Jedná se o základní knihovnu vláken pro operační systémy typu Unix (jako jsou Linux a macOS). Tato knihovna poskytuje velmi jemnou kontrolu pro správu vláken a synchronizačních prostředků. Protože se však jedná o nízkou úroveň abstrakce, jsou programy používající toto prostředí velmi náchylné k chybám, proto není vhodnou volbou pro vědecké výpočty, kde je klíčová přehlednost a srozumitelnost zdrojového kódu.

Programovací jazyk C++ prošel v posledních 15 letech významným vývojem, aby vyhověl požadavkům vícejádrových a multiprocesorových výpočetních systémů. Před verzí C++11 bylo nutné využít prostředí paralelního programování specifického pro konkrétní platformu, jako jsou Pthreads pro Unixové systémy a Windows Threading Library pro operační systémy Windows. Od verze C++11 poskytuje standardní knihovna jazyka C++ platformově nezávislý a objektově orientovaný způsob práce s vlákny včetně základních synchronizačních prostředků. V C++17 přibýly paralelní verze algoritmů z knihovny STL (v angl. *Standard Template Library*), a ve verzi C++20 pak přibýly pokročilé synchronizační prostředky a nová třída pro práci s vlákny, která splňuje principy RAII (zkr. pro *resource acquisition is initialization*). [11]

Vícevláknové programování v C++ již není tolik nízkoúrovňové, jako je tomu například v knihovně Pthreads, avšak stále může být potřeba provést velké změny ve zdrojovém kódu sekvenčního algoritmu, aby byl přepsán na paralelní verzi. Naproti tomu knihovna OpenMP poskytuje vysokou úroveň abstrakce pro paralelní programování a umožňuje paralelizaci sekvenčního algoritmu s poměrně malým množstvím změn ve zdrojovém kódu. Většina funkcionalit OpenMP je dostupná prostřednictvím direktiv pro překladač, zbylé funkcionality jsou pak přístupné prostřednictvím knihovnických funkcí. Díky své jednoduchosti a možnosti inkrementálního přidávání paralelního kódu se na konci devadesátých let stalo OpenMP dominantní volbou pro programování systémů se sdílenou pamětí. [10]

Protože se tato práce zabývá porovnáním paralelizačních přístupů pomocí C++ a OpenMP, budou obě tato prostředí paralelního programování podrobněji popsána.

3.1 Paralelní programování v C++

Vývoj počítačové architektury směrem k vícejádrovým a multiprocesorovým systémům zásadně změnil způsob, jakým musí být navržen software, aby dosáhl dobrého výkonu. Jazyk C++ na tuto změnu reagoval postupným zaváděním komplexních nástrojů pro paralelní programování od základních funkcionalit uvedených ve verzi C++11 po pokročilé synchronizační prostředky dostupné ve verzi C++20.

3.1.1 Paměťový model

Před verzí C++11 se standard jazyka C++ nezabýval vícevláknovým programováním a paralelní programování bylo přenecháno rozšířením specifickým pro danou platformu, jako je knihovna Pthreads pro Unixové systémy nebo Windows Threading Library pro operační systémy Windows. Protože C++11 přineslo podporu pro vícevláknové programování, bylo nutné zavést paměťový model, který formálně definuje, jak se chovají souběžné operace nad sdílenou pamětí a jak jsou tyto operace uspořádány mezi vlákny. [11, 12]

Paměťový model jazyka C++ je založen na konceptu paměťových oblastí a operacích, které k těmto oblastem přistupují. Paměťová oblast je objekt skalárního typu (aritmetický typ, typ ukazatele, výčtový typ a `std::nullptr_t`), nebo největší souvislá posloupnost bitových pozic nenulové délky. Každé čtení nebo zápis je pak paměťovou operací, která se vykonává nad specifickou paměťovou oblastí. [11]

Pro určení výsledku situace, kdy dvě vlákna současně přistupují ke stejné paměťové oblasti, používá paměťový model C++ pojem časově závislé chyby (v angl. *data race* nebo také *race condition*). Pokud dvě vlákna souběžně přistupují ke stejné paměťové oblasti, alespoň jedno z těchto vláken chce danou paměťovou oblast modifikovat a platí obě následující podmínky:

1. daná paměťová oblast není modifikována atomickou operací (viz. 3.1.4);
2. přístupy vláken nejsou v tzv. *happens-before* relaci (viz. dále),

pak takový program obsahuje časově závislou chybu a chování tohoto programu není definováno. Je potřeba zmínit, že pojem časově závislé chyby se sice opírá o termín souběžného přístupu vláken, avšak formální definování této souběžnosti není triviální úloha. [11, 12]

Protože paměťový model jazyka C++ považuje časově závislé chyby za ne-definované chování, může kompilátor předpokládat, že se v programu žádné takové chyby nevyskytují a provádět silné optimalizace zdrojového kódu, jako je např. přeuspořádání provedení operací. Za účelem zvýšení výkonu programu může operace přeskupovat kompilátor, optimalizátor i hardwarový plánovač instrukcí v CPU. Aby existovaly určité záruky, na které se lze přes všechny tyto optimalizace spolehnout, definuje paměťový model jazyka C++ dva koncepty, které programátorovi poskytují kontrolu nad přeuspořádáním operací. [11, 12]

Prvním tímto konceptem jsou *happens-before* relace. Tato relace mezi operacemi A a B zaručuje, že všechny efekty operace A jsou viditelné pro operaci B. Relace *happens-before* se skládá ze dvou druhů následujících relací:

- Relace *sequenced-before* určuje, které operace v rámci téhož vlákna nesmějí být přeuspořádány. [11]
- Relace *synchronizes-with* zaručuje pořadí provedení určitých operací mezi jednotlivými vlákny. Do této relace patří například následující operace:
 - Operace vytvoření nového vlákna se provede před provedením první operace v tomto nově vytvořeném vlákně. [11]
 - Poslední operace v daném vlákně se provede předtím, než se vrátí volání metody `join` (viz. 3.1.2) ve vlákně, které toto vlákno vytvořilo. [11]
 - Operace uvolnění zámku (viz. 3.1.3) v jednom vlákně se provede předtím, než se provede operace získání zámku v jiném vlákně. [11]

Relace happens-before je tranzitivní, tj. pokud A je v relaci s B a B je v relaci s C, pak je i operace A v relaci s operací C. [11]

Druhý koncept, který poskytuje kontrolu nad přeuspořádáním instrukcí, je výčtový typ `std::memory_order`. Pomocí tohoto typu je možné určit pořadí, v jakém budou operace čtení a zápisu v jednom vlákne viditelné pro ostatní vlákna. Tato možnost kontroly nad přeuspořádáním operací je unikátní pro jazyk C++ a není dostupná v paměťovém modelu jazyků C# nebo Java. [11]

Pro pochopení těchto uspořádání je klíčové, že nastavení daného uspořádání se vztahuje na pořadí paměťových operací v jednom konkrétním vlákne, tj. nastavení daného uspořádání v jednom vlákne neovlivňuje pořadí operací v ostatních vláknech. [11] Existuje celkem šest následujících uspořádání:

- **memory_order_seq_cst**: Model sekvenční konzistence, tj. je zaručeno, že existuje jedno sekvenční pořadí všech paměťových operací v daném vlákne, které je konzistentní s pořadím těchto operací ve zdrojovém kódu a je pozorovatelné pro všechna ostatní výpočetní vlákna. Toto je pro atomické datové typy výchozí uspořádání. [11, 12]
- **memory_order_relaxed**: Není zaručeno žádné uspořádání, tj. vlákno může pozorovat paměťové operace s **memory_order_relaxed** uspořádáním z jiného vlákna v jakémkoliv pořadí. [11, 12]
- **memory_order_release**: Zaručuje, že žádné operace pro čtení a zápis v aktuálním vlákne nesmí být přeskupeny po zápisu s **memory_order_release** uspořádáním. Tedy platí, že pokud jiná výpočetní vlákna vidí výsledek zápisu s uspořádáním **memory_order_release**, pak zaručeně vidí i efekty veškerých předchozích paměťových operací z daného vlákna. [11]
- **memory_order_acquire**: Zaručuje, že žádné operace pro čtení a zápis v aktuálním vlákne nesmí být přeskupeny před čtením s tímto paměťovým uspořádáním. [11, 12]
- **memory_order_consume**: Toto uspořádání je volnější formou uspořádání předchozího. U operace s **memory_order_consume** uspořádáním všechna ostatní vlákna vidí efekty této operace před efekty jakékoliv následující paměťové operace, avšak s výjimkou toho, že efekty, které nejsou závislé na hodnotě operace s **memory_order_consume** uspořádáním, mohou nastat ještě před touto operací. [12]
- **memory_order_acq_rel**: Toto uspořádání kombinuje sémantiku uspořádání **memory_order_acquire** a **memory_order_release**. [11]

Paměťový model jazyka C++ je velice komplexní a hluboké téma, které přesahuje rozsah této práce. Avšak více o paměťovém modelu jazyka C++ se lze dočíst např. v [11] nebo [12].

3.1.2 Správa vláken

Výpočet, který může být potenciálně prováděn souběžně s jinými výpočty, se nazývá úloha. Vláknem je systémovou reprezentací úlohy v programu. Úloha, která má být spuštěna souběžně s jinými úlohami, se spustí vytvořením instance třídy `std::thread` s danou úlohou jako argumentem konstruktoru této třídy. Úloha může být reprezentována jako funkce, funkční objekt nebo lambda funkce. [11, 12]

Životnost vytvořeného vlákna končí s vykonáním úlohy, která mu byla předána, přičemž na každém `std::thread` objektu musí být zavolána metoda `join` nebo `detach` před tím, než dojde k jeho destrukci. Volání `join` je blokující, dokud volané vlákno nedokončí svůj výpočet. Volání `detach` odděluje výpočet daného vlákna od jeho `std::thread` objektu, čímž mu umožňuje provádět svůj výpočet nezávisle na vlákně, které jej vytvořilo. V případě odpojení vlákna pomocí volání `detach` se však může stát, že hlavní vlákno dokončí svůj výpočet dříve než toto odpojené vlákno. Pokud na objektu vytvořeného vlákna není žádná z těchto dvou metod zavolána před jeho destrukcí, pak je v destruktoru tohoto vlákna automaticky ukončen celý program prostřednictvím volání `std::terminate`. [11, 12]

Proměnná, která je označena jako `thread_local`, je vlastněná daným vláknem, tj. každé vlákno má svou vlastní instanci této proměnné. Je však nutné si uvědomit, že ačkoliv je taková proměnná v daném vlákně nepřístupná pro zbylá vlákna, mohou ostatní vlákna k této proměnné přistoupit v případě, že vlákno, které tuto proměnnou vlastní, poskytne její adresu prostřednictvím ukazatele, neboť virtuální paměťový prostor procesu je sdílený mezi všemi výpočetními vlákny (viz. 2.3). Příklad použití vláken ukazuje výpis 3.1. [12]

Ve verzi C++20 přibyla třída `std::jthread`, která přináší dvě zásadní vylepšení. Prvním vylepšením je, že v destrukturu této třídy je automaticky zavolána metoda `join`, takže již není potřeba ji na tomto objektu volat explicitně. Tím tato třída splňuje princip RAII, který říká, že životnost objektu je svázána s rozsahem platnosti jeho proměnné, tj. když skončí rozsah platnosti proměnné, musí být v jejím destrukturu uvolněn zdroj, který tato proměnná spravuje. Díky tomuto mechanismu je na objektu této třídy zavolána metoda `join` i v případě, že během výpočtu nastane výjimka. [11]

Druhým vylepšením je možnost kooperativního ukončení výpočtu vlákna. Násilné ukončení vlákna není bezpečné, protože nemusí být znám stav výpočtu tohoto vlákna. Konstruktory třídy `std::jthread` umožňuje přijmout funkci, která jako první argument přijímá objekt typu `std::stop_token`. Prostřednictvím tohoto objektu může vlákno ověřovat, jestli bylo požádáno o zastavení a bezpečně tak ukončit svůj výpočet. [11]

3.1.3 Vzájemné vyloučení

Třída `std::mutex` nabízí zajištění vzájemného vyloučení. Jedná se tedy o typ zámku, který zaručuje, že ke kritické sekci bude mít přístup vždy pouze jedno vlákno. Uzamknutí se provádí blokující operací `lock`, její neblokující varianta je `try_lock`. Uvolnění zámku se pak provádí metodou `unlock`. [11] Dalšími typy zámků jsou:

- `std::recursive_mutex`: Tento zámek umožňuje danému vláknům zamknout objekt více než jednou. Tento zámek zůstává zamčený, dokud není odemčen tolikrát, kolikrát byl uzamčen. [11]
- `std::timed_mutex`: Tato třída umožňuje pomocí operací `try_lock_for` a `try_lock_until` blokovat vlákno při pokusu o získání zámku pouze po určité době. Pokud se vláknům nepodaří zámek získat ve stanoveném časovém limitu, tak se pokus o získání zámku ukončí. [11]
- `std::recursive_timed_mutex`: Tato třída kombinuje vlastnosti předchozích dvou zámků.
- `std::shared_timed_mutex`: Tato třída byla přidána ve verzi C++14 a poskytuje možnost použít instanci této třídy pro exkluzivní i sdílené zamykání. Jedno vlákno může objekt uzamknout exkluzivně pro účely modifikace dat, zatímco více vláken může zamknout objekt pro souběžné čtení dat. Zároveň tato třída umožňuje podobně jako `std::timed_mutex` zamykání s časovými intervaly. [11]
- `std::shared_mutex`: Tato třída přibyla ve verzi C++17 a poskytuje stejnou funkcionalitu jako předchozí, avšak bez podpory časového limitu pokusu o získání zámku. [11]

Ruční zamykání a odemykání objektů zajišťující vzájemné vyloučení je však náchylné k chybám typu uváznutí (v angl. *deadlock*), kdy jedno nebo více vláken čeká na uvolnění zámku, který však např. pokud nastane výjimka nikdy nebude uvolněn. Z těchto důvodů C++ poskytuje třídy pro automatickou správu zámků, které riziko uváznutí významně snižují. Tyto třídy splňují princip RAII, neboť zámek si přivlastňují a případně jej rovnou zamykají v konstruktoru a automaticky jej uvolňují ve svém destrukturu. [11] Mezi tyto třídy patří:

- `std::lock_guard`: Tato třída poskytuje základní funkcionalitu pro automatickou správu zámku, tj. uzamknutí zámku v konstruktoru a jeho uvolnění ve svém destrukturu. [11]
- `std::unique_lock`: Oproti `std::lock_guard` poskytuje tato třída rozšířenou funkcionalitu, jako je např. odložené uzamknutí získaného zámku, manuální zamykání a odemykání nebo přesunutí vlastnictví zámku prostřednictvím `move` sémantiky. [11]

- `std::shared_lock`: Tato třída je dostupná od C++14 a poskytuje automatickou správu zámků `std::shared_timed_mutex` a `std::shared_mutex` pro získání sdíleného přístupu ke kritické sekci mezi více vlákny. [11]
- `std::scoped_lock`: Od C++11 je k dispozici funkce `std::lock`, prostřednictvím které je možné uzamknout dva a více zámků atomicky pomocí algoritmu pro zabránění uváznutí. Třída `std::scoped_lock`, která je dostupná od C++17, pak kombinuje tuto funkcionalitu s automatickou správou zámků. [11]

Příklad použití vzájemného vyloučení ukazuje výpis 3.1. Tento výpis zároveň demonstruje rozdílné použití tříd pro automatickou správu zámků. Pro odesílatele notifikace je postačující `std::lock_guard`, neboť je zámek zamknut a odemknut pouze jedenkrát. Avšak příjemce musí použít `std::unique_lock`, protože může zámek uzamknout a odemknout vícekrát. [11]

3.1.4 Atomické typy

Atomický typ je specializace šablony `std::atomic<T>`. Operace nad atomickým datovým typem je nedělitelná, tj. je provedena jedním vláknem bez přerušování od ostatních vláken. U většiny implementací navíc platí, že pro aritmetické typy a typ ukazatele jsou všechny atomické operace nad těmito typy implementovány bez zámků. Operace `load` a `store` slouží k načítání a ukládání hodnoty do atomické proměnné a současně umožňují zvolit jedno z paměťových uspořádání, která byla popsána v 3.1.1. Od C++20 atomické typy také umožňují blokovat dané vlákno voláním metody `wait`, dokud hodnota atomické proměnné není rovna požadované hodnotě, a notifikovat ostatní vlákna při změně hodnoty této proměnné pomocí metod `notify_one` a `notify_all`. Tato funkcionalita je velmi podobná podmíněným proměnným, které jsou popsány v kapitole 3.1.5. [11, 12]

C++20 přidalo možnost používat atomické chytré ukazatele. Chytrý ukazatel `std::shared_ptr` se skládá z řídicího bloku a z ukazatele, který spravuje. Řídicí blok je bezpečný pro souběžný přístup více vláken, tj. operace s čítačem počítaných referencí jsou atomické a současně je zaručeno, že spravovaný objekt bude uvolněn právě jednou. Nicméně přístup k samotnému spravovanému ukazateli již není bezpečný pro souběžný přístup. Z těchto důvodů ve standardu C++20 přibýly atomické chytré ukazatele v podobě specializací `std::atomic<std::shared_ptr>` a `std::atomic<std::weak_ptr>`. [11]

Ve verzi C++20 byly také uvedeny atomické reference `std::atomic_ref`. Tato třída umožňuje aplikovat atomické operace na referencovaný objekt, proto souběžné zápisy a čtení na tomto objektu nepředstavují časově závislou chybu. Je však nutné, aby životnost referencovaného objektu překračovala životnost instance `std::atomic_ref`. Navíc, pokud je pro přístup k objektu použita atomická reference, všechny ostatní přístupy k objektu musí taktéž použít atomickou referenci. [11]

3.1.5 Podmíněné proměnné

Podmíněné proměnné `std::condition_variable` se používají ke správě komunikace mezi jednotlivými vlákny. Vláknem může čekat (být blokováno) na podmíněné proměnné voláním metody `wait`, dokud nebude splněna určitá podmínka. Metoda `wait` podmíněné proměnné očekává jako první argument objekt typu `std::unique_lock<std::mutex>`. Existuje také přetížená varianta, která jako druhý argument přijímá predikát reprezentující podmínku, která se má při čekání nebo po probuzení ověřovat. Podmíněná proměnná tak představuje efektivnější alternativu k aktivnímu čekání, které zbytečně plýtvá systémovými prostředky. Prostřednictvím podmíněné proměnné si tedy mohou jednotlivá vlákna posílat notifikace pomocí volání `notify_one`, které probudí právě jedno čekající vláknem, nebo `notify_all`, které probudí všechna čekající vlákna. [11, 12]

Používání podmíněných proměnných je náchylné ke ztracenému probuzení (v angl. *lost wakeup*) a falešnému probuzení (v angl. *spurious wakeup*). Ztracené probuzení nastává v případě, že odesílatel odešle notifikaci dříve, než se příjemce dostane do stavu čekání na notifikaci. Tento jev lze však řešit metodou čekání s predikátem. Falešné probuzení nastává, když je čekající vláknem probuzeno, ačkoliv notifikace nebyla žádným jiným vláknem odeslána. Toto chování lze pozorovat na některých systémech, které povolují falešné probuzení za účelem snazší implementace podmíněných proměnných. [12]

Příklad použití podmíněné proměnné ukazuje výpis kódu 3.1. V této ukázce musí příjemce použít `std::unique_lock` nejen proto, že zámek může příjemce uzamknout a odemknout vícekrát, pokud se k podmíněné proměnné dostane dříve než odesílatel notifikace, ale také kvůli riziku falešného probuzení. [11]

3.1.6 Paralelní algoritmy v STL

C++17 přineslo významné vylepšení v podobě možnosti paralelizace většiny algoritmů ze standardní knihovny STL. Od této verze je možné algoritmy spouštět s tzv. způsobem provádění (z angl. *execution policy*), což umožňuje paralelizaci existujících řešení s minimálními změnami ve zdrojovém kódu. Pomocí způsobu provádění lze určit, zda se má daný algoritmus spustit sekvenčně, paralelně nebo paralelně s vektorizací. [11, 13] Hlavičkový soubor `<execution>` definuje celkem čtyři způsoby provádění:

- `std::execution::seq`: Sekvenční spuštění v jednom vlákně, které je ekvivalentní tradičnímu spuštění daného algoritmu. [13]
- `std::execution::par`: Povoluje paralelní spuštění algoritmu ve více výpočetních vláknech. [13]
- `std::execution::par_unseq`: Povoluje paralelní spuštění algoritmu ve více vláknech, přičemž výpočet může být vektorizován, tj. zpracovávat více datových položek souběžně. [13]

```
#include <iostream>
#include <condition_variable>
#include <mutex>
#include <thread>

std::mutex mutex_;

std::condition_variable condVar;

bool dataReady{false};

void doTheWork() {
    std::cout << "Processing shared data." << std::endl;
}

void waitingForWork() {
    std::cout << "Worker: Waiting for work." << std::endl;
    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}

void setDataReady() {
    {
        std::lock_guard<std::mutex> lck(mutex_);
        dataReady = true;
    }
    std::cout << "Sender: Data is ready." << std::endl;
    condVar.notify_one();
}

int main() {

    std::thread t1(waitingForWork);
    std::thread t2(setDataReady);

    t1.join();
    t2.join();

    return 0;
}
```

■ **Výpis kódu 3.1** Ukázka použití vláken, zámku a podmíněné proměnné. Zdrojový kód převzat z [11].

- `std::execution::unseq`: Tato možnost byla přidána v C++20 a povoluje vektorizovaný výpočet, avšak bez paralelizace ve více vláknech. [13]

Je důležité poznamenat, že specifikováním způsobu provádění není zaručeno, že daný algoritmus skutečně bude proveden paralelně. Programátor tímto způsobem pouze vyjadřuje, že upřednostňuje, aby byl algoritmus spuštěn paralelně. Nicméně algoritmus může být spuštěn sekvenčně, pokud hardware tento způsob provádění nepodporuje, nebo pokud se implementace ze strategických důvodů rozhodne pro sekvenční spuštění, např. protože zátěž všech výpočetních jader je ještě před spuštěním daného algoritmu příliš vysoká. [11, 13]

Od verze C++17 je paralelní spuštění podporováno v 69 algoritmech z STL a 8 nových algoritmů, které podporují paralelní výpočet, bylo přidáno. Algoritmus `std::for_each` aplikuje unární funkci na každý prvek z daného rozsahu, `std::for_each_n` pak aplikuje unární funkci na prvních n prvků od začátku poskytnutého rozsahu. Algoritmus redukce `std::reduce` poskytuje alternativu k `std::accumulate`, která nezaručuje pořadí vyhodnocování zleva doprava. Kvůli paralelnímu provádění algoritmus redukce vyžaduje, aby poskytnutá operace byla asociativní i komutativní. Operace `std::transform_reduce` kombinuje funkcionalitu algoritmů `std::transform` a `std::reduce`, tj. nejprve použije poskytnutou unární funkci na daný rozsah, příp. binární funkci na dva rozsahy, a na tento výsledný rozsah pak aplikuje redukci. Operace `std::transform_reduce` proto reprezentuje implementaci známého algoritmu MapReduce. Další dva algoritmy pak implementují prefixové součty, konkrétně `std::inclusive_scan` implementuje inkluzivní prefixový součet a operace `std::exclusive_scan` exkluzivní prefixový součet. Ze stejných důvodů jako v případě paralelní redukce vyžadují oba tyto algoritmy, aby poskytnutá binární operace byla asociativní. Poslední dva algoritmy přidané v C++17 jsou `std::transform_inclusive_scan` a `std::transform_exclusive_scan`, jež kombinují funkcionalitu transformace a prefixového součtu.[11]

3.1.7 Pokročilé synchronizační prostředky

C++20 přidalo možnost používat pokročilé synchronizační prostředky, konkrétně byly přidány závory, bariéry a semaforey. Závora `std::latch` je jednorázový synchronizační prostředek, který umožňuje vláknům čekat tak dlouho, dokud čítač nedosáhne nuly. Po nastavení počáteční hodnoty čítače mohou různá vlákna atomicky snižovat tuto hodnotu, přičemž když čítač dosáhne nuly, tak všechna vlákna, která na závoře čekají, jsou probuzena a pokračují ve výpočtu. Závoru nelze použít opakovaně, neboť jakmile čítač dosáhne nuly, tak se již neresetuje na původní hodnotu. [11, 13]

Bariéra `std::barrier` rozšiřuje funkcionalitu závory. Zprv je možné bariéru používat opakovaně, neboť po snížení čítače na nulu a probuzení čekajících vláken je čítač resetován na původní hodnotu. Tato opakovaná použití bariéry se také označují jako fáze. Zadruhé je možné snížit hodnotu čítače pro

současnou a zároveň i příští fázi bariéry. Dále bariéra umožňuje na konci každé fáze spuštění funkce, která byla bariéře předána v jejím konstruktoru. Když čítač bariéry dosáhne nulové hodnoty, je jedno z čekajících vláken probuzeno, aby tuto funkci vykonalo. Teprve poté, co probuzené vlákno úspěšně dokončí provádění této funkce, jsou probuzena i ostatní čekající vlákna a všem těmto vláknům je pak povoleno pokračovat ve výpočtu. [11, 13]

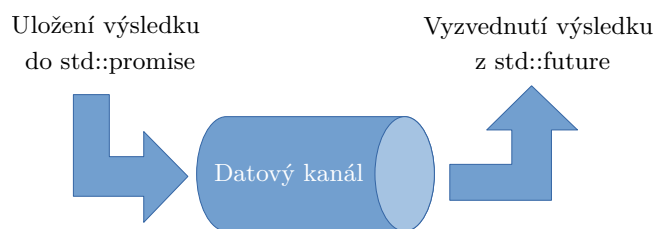
Posledním synchronizačním prostředkem, který je dostupný od C++20, jsou semaforey¹. Semafor je synchronizační mechanismus, který se používá k řízení souběžného přístupu k omezenému počtu výpočetních prostředků. Konstruktor semaforu `std::counting_semaphore` přijímá počáteční hodnotu čítače, přičemž vlákno, které chce semafor získat voláním `acquire`, hodnotu čítače atomicky sníží, nebo je zablokováno, pokud čítač dosáhl nulové hodnoty. Volání metody `release` hodnotu čítače semaforu naopak atomicky zvyšuje, případně toto volání probudí jedno z čekajících vláken. Třída `std::binary_semaphore` má jako maximální hodnotu interního čítače jedničku, proto je tento semafor možné využít také jako zámek, ovšem s výhodou, že vlákno uvolňující semafor nemusí být stejné vlákno, které semafor předtím získalo. [11, 13]

3.1.8 Asynchronní programování

C++11 přineslo podporu pro asynchronní programování, známé z jiných programovacích jazyků, prostřednictvím funkční šablony `std::async`, která umožňuje vytvořit tzv. úlohu (z angl. *task*). Zatímco vlákna vyžadují pro komunikaci sdílené proměnné, které musí být chráněny zámekem, úloha komunikuje prostřednictvím datového kanálu bez explicitního použití zámku. Koncové body tohoto datového kanálu se nazývají `std::promise` a `std::future`, který je návratovou hodnotou volání `std::async`. Objekt `std::future` reprezentuje hodnotu, která bude k dispozici v budoucnu, a `std::promise` poskytuje mechanismus pro nastavení této hodnoty. Metoda `get` objektu `std::future` zablokuje volající vlákno, dokud nebude výsledek úlohy k dispozici. Výhodou tohoto přístupu je automatická propagace výjimek, tj. do objektu `std::promise` je možné uložit jak výsledek úspěšného výpočtu, tak i výjimku, která je pak automaticky propagována do vyzvednutí výsledku při volání metody `get` na `std::future`. Ilustraci tohoto datového kanálu ukazuje obrázek 3.1 [11, 12]

Zatímco `std::thread` explicitně vytvoří nové vlákno, které má zpracovávat zadanou funkci, úloha `std::async` může potenciálně být spuštěna v separátním vlákně. Runtime jazyka C++ rozhodne, zda má být úloha spuštěna ve stejném, nebo v separátním vlákně, přičemž toto rozhodnutí může záviset na počtu dostupných jader CPU nebo aktuálním zatížení výpočetního systému. Alternativně je možné specifikovat, zda má být úloha spuštěna v novém nebo ve stávajícím vlákně pomocí politiky spuštění. Politika `std::launch::async` zaručuje spuštění v novém vlákně, zatímco `std::launch::deferred` odkládá

¹Koncept semaforu vynalezl v roce 1965 Edsger W. Dijkstra. [11]



■ **Obrázek 3.1** Úloha jako datový kanál mezi dvěma komunikačními body. Překresleno z [11].

spuštění výpočtu až do okamžiku, kdy je na `std::future` zavoláno vyzvednutí výsledku, čímž je daná úloha spuštěna synchronně ve volajícím vlákne. Protože úloha může být spuštěna ve stejném nebo separátním vlákne, tak i oba koncové body datového kanálu `std::promise` a `std::future` mohou existovat v jednom nebo ve dvou různých vláknech. Pokud je potřeba oddělit vytvoření úlohy od jejího spuštění, je možné použít `std::packaged_task`, který umožňuje úlohu definovat a samotné spuštění odložit na později. Objekt `std::packaged_task` je navíc na rozdíl od `std::async` možné resetovat a použít znovu. [11]

Další významná podpora asynchronního programování byla do jazyka přidána ve verzi C++20, která přinesla korutiny² (v angl. *coroutines*). Korutiny jsou funkce, které jsou schopny pozastavit a obnovit své provádění, přičemž si při pozastavení zachovávají svůj vnitřní stav výpočtu. Zatímco standardní funkce se po zavolání vykonávají od začátku do konce, výpočet korutin může být rozdělen do několika dílčích částí, kdy je možné v určitých okamžicích výpočet korutiny pozastavit, dokud nebude obnoven. Korutina může být pozastavena, protože např. musí čekat na určitou událost, nebo má k dispozici mezivýsledek výpočtu, který chce předat volajícímu. Toto chování může připomínat posloupnost volání několika funkcí, nicméně klíčový rozdíl je, že korutina si zachovává svůj stav od okamžiku, kdy byla pozastavena, až do chvíle, kdy je obnovena. C++20 přineslo podporu pro definování vlastních korutin, pro které je navíc nezbytné definovat také vlastní návratový typ. Ve verzi C++23 je k dispozici první konkrétní návratový typ `std::generator`, který lze použít pro korutiny generující potenciálně nekonečnou posloupnost prvků. [11, 13]

3.2 OpenMP

OpenMP je specifikace API určená pro paralelní programování v jazycích C, C++ a Fortran. Tato specifikace je pak implementována různými překladači, jako je např. GCC, Clang nebo Microsoft Visual C++. Specifikace OpenMP se skládá z direktiv pro překladač, knihovních funkcí a proměnných prostředí. Direktiva OpenMP je instrukce ve speciálním formátu, kterou rozpoznávají

²Korutiny vynalezl v roce 1958 Melvin Conway. [13]

pouze kompilátory implementující OpenMP. Překladače, které OpenMP podporují, většinou také obsahují přepínač pro příkazovou řádku, který aktivuje interpretaci všech direktiv OpenMP. Díky tomuto přístupu je možné efektivně přepínat mezi sekvenční a paralelní kompilací programu, protože bez tohoto přepínače nejsou direktivy OpenMP překladačem rozeznány a jsou proto ignorovány. Zdrojový kód využívající pouze direktivy OpenMP se bez použití tohoto přepínače tedy zkompile jako standardní sekvenční kód. [14, 15]

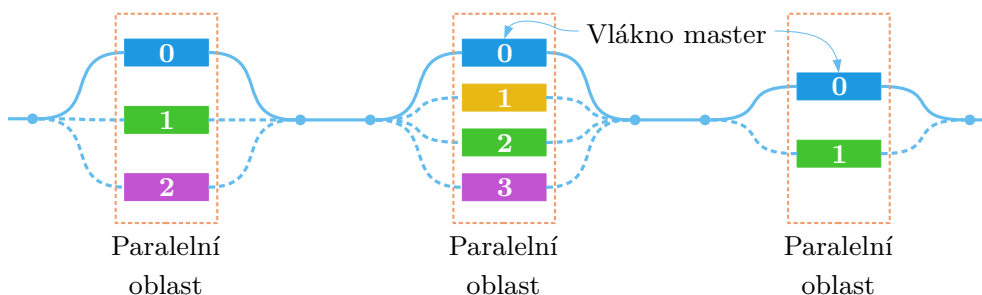
Funkcionality popsané v této kapitole pocházejí z OpenMP verze 5.0, která je k dispozici na výpočetním ARM uzlu fakultního klastru ClusterFIT, na kterém budou prováděna všechna měření v rámci této práce. Úplná specifikace OpenMP je nesmírně rozsáhlá a kompletní popis všech funkcionalit významně přesahuje rozsah této práce. Z těchto důvodů budou v této části popsány pouze nejdůležitější konstrukty. Zároveň se popsané funkcionality vztahují pouze na jazyk C++ a v programovacím jazyce Fortran se mohou některé vlastnosti mírně lišit.

3.2.1 Paměťový model

OpenMP používá jiný paměťový model sdílené paměti než jazyk C++, konkrétně model volnější konzistence (v angl. *relaxed-consistency model*). V tomto OpenMP modelu mají všechna vlákna přístup ke sdílené paměti, avšak kromě toho je každému vláknu umožněno uchovávat si svůj vlastní dočasný pohled na tuto sdílenou paměť. Tento dočasný pohled sice není specifikací OpenMP vyžadován, ale umožňuje reprezentovat jakýkoliv mezistupeň paměťové hierarchie, jako jsou registry procesoru, cache paměť nebo jiný druh lokální paměti mezi vláknem a hlavní pamětí. [14]

Dočasný pohled na sdílenou paměť umožňuje vláknům udržovat lokální kopie sdílených proměnných a tím se vyhnout načítání hodnot z hlavní paměti při každém jejich použití. Obdobně čtení hodnoty dané proměnné může načíst hodnotu z dočasného pohledu vlákna, pokud není vynuceno načtení ze sdílené paměti. Pokud je zapotřebí, aby všechna vlákna operovala s identickým obsahem sdílené paměti, pak musí programátor explicitně donutit příslušná vlákna, např. pomocí operace `flush`, k propsání lokálních hodnot sdílených proměnných do sdílené paměti, resp. k načtení aktuálních hodnot ze sdílené paměti do lokálních pamětí vláken. Dále má každé vlákno přístup ke své privátní paměti, ke které ostatní vlákna nesmí přistupovat. [14]

Podobně jako v případě jazyka C++ je i paměťový model OpenMP velmi komplexním tématem. Více o paměťovém modelu OpenMP, operaci `flush` a jejím vlivu na paměťovou konzistenci je možné se dočíst např. v oficiální OpenMP specifikaci [14].



■ **Obrázek 3.2** Ukázka fork-join modelu. Inspirováno ilustrací z [16].

3.2.2 Programovací model

OpenMP využívá tzv. *fork-join* programovací model, někdy také označovaný jako exekuční model (v angl. *execution model*). Program začne sekvenčně vykonávat jedno vlákno, které je ve specifikaci OpenMP označováno jako počáteční vlákno (v angl. *initial thread*). V explicitně označených místech zdrojového kódu, které se nazývají paralelní oblasti (z angl. *parallel regions*), vytvoří počáteční vlákno tým paralelně běžících vláken (v angl. *thread team*), a vlákna z tohoto týmu vykonávají kód této paralelní oblasti souběžně. Velikost týmu vláken a jeho složení se po dobu vykonávání dané paralelní oblasti nemění. Na konci každé paralelní oblasti je implicitní synchronizační bod v podobě bariéry, po níž pokračuje ve výpočtu pouze počáteční vlákno a výpočet ostatních vláken je ukončen. [14, 15]

Tento model umožňuje vnořený paralelismus, kdy vlákno v paralelní oblasti se samo může setkat s další paralelní oblastí a vytvořit tak další tým vláken. Vlákno, které se setká s paralelní oblastí a vytvoří nový tým vláken, se stává v rámci toho týmu hlavním vláknem (v angl. *master thread*) a dostává číselný index 0, přičemž zbylá vlákna tohoto týmu jsou číslována vzestupně od 1. Tím se OpenMP zásadně liší např. od C++, kde jsou číselné indexy všech vláken unikátní, zatímco v OpenMP se číselné indexy vztahují pouze na vlákna v jednom konkrétním týmu a různá vlákna z různých týmů tedy mohou mít stejné číselné indexy. Maximální hloubku zanoření vnořených paralelních oblastí je možné nastavit voláním funkce `omp_set_max_active_levels`. [14, 15] Ilustraci fork-join modelu ukazuje obrázek 3.2.

Direktivy OpenMP jsou speciální anotace, které překladačům předepisují generovat paralelní kód. Obecnou syntaxi direktiv OpenMP v jazyce C++ ukazuje výpis 3.2. Část `#pragma omp` označuje direktivu jako OpenMP konstrukci. Název direktivy určuje typ paralelní konstrukce, za kterým následují volitelné klauzule, pomocí kterých lze upravit chování této paralelní konstrukce. Za direktivou OpenMP pak musí následovat příkaz nebo strukturovaný blok kódu, na který se má konstrukce vztahovat. [14, 15]

```
#pragma omp direktiva [klauzule[[,] klauzule]...]
```

■ **Výpis kódu 3.2** Obecná syntaxe direktiv OpenMP v jazyce C++. [15]

3.2.3 Paralelní oblasti

Základní OpenMP konstrukce je direktiva `parallel`, která definuje paralelní oblast. Když se vlákno při výpočtu setká s touto konstrukcí, vytvoří nový tým vláken a všechna vlákna z tohoto týmu budou souběžně provádět příkaz nebo strukturovaný blok následující po této direktivě. [14]

Počet vláken v týmu a jejich chování je možné řídit různými klauzulemi, mezi které patří například následující klauzule:

- `if(condition)`: Tato klauzule poskytuje podmíněnou paralelizaci, tj. pokud je podmínka *condition* vyhodnocena jako nepravdivá, vykoná se daná paralelní oblast pouze jedním vláknem. V takovém případě je tedy tato paralelní oblast vykonána sekvenčně. Pokud je podmínka vyhodnocena jako pravdivá, dojde k vytvoření nového týmu vláken a kód paralelní oblasti bude prováděn paralelně tímto týmem. [15]
- `num_threads(expression)`: Pomocí této klauzule a *expression* výrazu lze určit počet vláken, která by měla být v týmu vykonávajícím paralelní oblast. Počet vláken v paralelních oblastech lze také nastavit globálně pomocí funkce `omp_set_num_threads` nebo proměnné prostředí `OMP_NUM_THREADS`, nicméně klauzule `num_threads` má vyšší prioritu než tyto zbylé dva způsoby. [14, 15]
- `private(list)`: Každé vlákno bude mít v této paralelní oblasti k dispozici svou vlastní lokální kopii každé proměnné ze seznamu *list*. Každá tato proměnná má na začátku paralelní oblasti nedefinovanou hodnotu a její hodnota je po skončení paralelní oblasti také nedefinována, tj. původní hodnota proměnné před paralelní oblastí není zachována. [15]
- `firstprivate(list)`: Podobné jako klauzule `private`, ale kopie proměnné každého vlákna je inicializována na hodnotu původní proměnné před paralelní oblastí. [15]
- `shared(list)`: Deklaruje všechny proměnné ze seznamu *list* jako sdílené mezi všemi vlákny této paralelní oblasti. Přístup ke sdíleným proměnným typicky vyžaduje použití synchronizačních prostředků, aby se zabránilo časově závislým chybám. [15]
- `default(option)`: Specifikuje prostřednictvím volby *option* výchozí vlastnost všech proměnných, které jsou použité v dané paralelní oblasti. Volba `shared` určuje, že použité proměnné jsou ve výchozím nastavení sdílené

mezi vlákny, a volba **none** určuje, že je potřeba pro každou použitou proměnnou explicitně specifikovat její atribut pro sdílení. [14, 15]

- **reduction(*operator: list*)**: Provádí redukci s operací **operator** na proměnných ze seznamu **list** a kombinuje hodnoty lokální pro každé vlákno do jediného výsledku. Lokální proměnné jsou inicializovány na neutrální prvek vůči zadané operaci a zkombinované lokální hodnoty se na závěr zkombinují s hodnotu proměnné před paralelní oblastí a tento celkový výsledek je zapsán do původní proměnné. [15]
- **copyin(*list*)**: Inicializuje proměnnou ze seznamu **list** hodnotou z počátečního vlákna. Každá proměnná v tomto seznamu musí být deklarována jako **threadprivate**, viz. dále. [14, 15]

Pokud je potřeba, aby si proměnná uchovávala svou hodnotu napříč různými paralelními oblastmi, je možné použít direktivu **threadprivate(list)**. Při použití této direktivy bude mít každá proměnná ze seznamu **list** svou lokální kopii v každém vlákně, přičemž tato kopie je udržována napříč všemi paralelními oblastmi, tj. hodnota této proměnné je zachována při přestupu mezi paralelními oblastmi. Pro použití této direktivy je nutné, aby ve všech paralelních oblastech byla velikost týmů vláken stejná. Pro účely definování hodnoty proměnné ze seznamu **list** na začátku první paralelní oblasti musí být v deklaraci této paralelní oblasti použita klauzule **copyin**. [14, 15]

3.2.4 Datový paralelismus

OpenMP podporuje dva základní modely pro vyjádření paralelismu. Prvním modelem je datový paralelismus (v angl. *data parallelism*), ve kterém se data distribuují mezi více vláken, přičemž každé vlákno provádí stejné operace na různých částech dat. Datový paralelismus realizuje direktiva **for**, která rozděluje iterace smyčky mezi vlákna týmu v dané paralelní oblasti. Na konci těla paralelního cyklu je podobně jako u paralelní oblasti implicitní synchronizační bariéra. Tuto direktivu je možné zkombinovat s paralelní direktivou do jednoho řádku pomocí konstrukce **parallel for**. [15] Direktiva **for** podporuje následující klauzule:

- **collapse(*n*)**: Tato klauzule umožňuje paralelizovat více úrovní smyček. Pokud je parametr $n \geq 1$, pak jsou iterace přidružených smyček, na které se klauzule vztahuje, sdruženy do jedné větší iterační smyčky, která je poté rozdělena podle klauzule **schedule**, viz. dále. [14, 15]
- **nowait**: Pokud je uvedena tato klauzule, je odstraněna implicitní bariéra na konci těla paralelního cyklu, tj. vlákna po dokončení svých iterací nečekají na bariéry a rovnou pokračují v dalším výpočtu. [15]

- `schedule(type[, chunk_size])`: Klauzule určuje způsob, kterým jsou iterace smyčky rozděleny mezi výpočetní vlákna. Mezi způsoby rozdělení, které jsou určeny parametrem `type`, patří:
 - `static`: Iterace jsou rozděleny do bloků o velikosti `chunk_size`. Bloky jsou přidělovány vláknům staticky v pořadí podle čísla vlákna cyklickým způsobem, přičemž poslední přiřazený blok může mít menší počet iterací. Pokud velikost bloku není specifikována, jsou iterace rozděleny na bloky přibližně stejné velikosti. Každému vláknu je pak přiřazen maximálně jeden blok. [15]
 - `dynamic`: Iterace jsou vláknům přidělovány dynamicky na vyžádání. Vláknem provede blok iterací, jehož velikost určuje parametr `chunk_size`, a poté požádá o další blok, dokud nezbyvají žádné další bloky ke zpracování. Poslední blok může mít méně iterací, než je specifikovaná velikost. Pokud není velikost bloku zadána, je výchozí hodnota rovna 1. [15]
 - `guided`: Stejně jako v předchozím případě jsou bloky přidělovány vláknům dynamicky, liší se pouze velikosti přidělovaných bloků. Pokud je `chunk_size = 1`, je velikost každého bloku úměrná počtu dosud nepřijížených iterací dělenému počtem vláken v týmu. Pro `chunk_size` větší než 1 se velikost každého bloku určuje stejným způsobem, avšak s omezením, že bloky neobsahují méně než `chunk_size` iterací, s výjimkou posledního zbytkového bloku iterací. [15]
 - `auto`: Rozhodnutí o způsob přidělení iterací vláknům je delegováno na kompilátor či běhové prostředí. [14]
 - `runtime`: Způsob přidělování iterací je určen v době běhu programu prostřednictvím proměnné prostředí `OMP_SCHEDULE`. [15]

Stejně jako u paralelních oblastí je pro direktivu `for` dostupná klauzule `reduction` a podporované jsou také klauzule pro deklaraci vlastností proměnných, jako jsou `private`, `firstprivate` a `shared`. Direktiva `for` navíc umožňuje použití klauzule `lastprivate(list)`. Tato klauzule zajišťuje, že proměnná uvedená v seznamu `list` je přístupná po dokončení provedení `for` direktivy a bude mít hodnotu z iterace smyčky, která by byla poslední v sekvěním provedení. [15]

3.2.5 Úlohový paralelismus

Druhým modelem vyjádření paralelismu v OpenMP je úlohový paralelismus (z angl. *task parallelism* nebo také *functional parallelism*). Tento druh paralelismu zahrnuje rozdělení různých úloh mezi výpočetní vlákna, přičemž každé vláknem může provádět odlišné operace. Úlohový paralelismus zajišťuje direktiva `task`, která umožňuje explicitní vytvoření úlohy jako jednotky paralelního výpočtu. [14, 15]

Když vlákno potká při výpočtu tuto direktivu, je z kódu pro přidružený strukturovaný blok zdrojového kódu vygenerována explicitní úloha, kterou může toto vlákno okamžitě provést nebo její provedení odložit. Odložené úlohy pak mohou být dynamicky přiřazeny libovolnému vláknu v daném týmu. Vygenerovaná úloha se někdy nazývá synovská úloha (v angl. *child task*) a úloha, která vygeneruje novou explicitní úlohu, se také nazývá aktuální úloha (v angl. *current task*). [14] Mezi hlavní klauzule direktivy **task** patří následující klauzule:

- **if(*condition*)**: Pokud je přítomna tato klauzule a podmínka *condition* je vyhodnocena jako nepravdivá, vygeneruje se nová úloha a vlákno, které se s touto direktivou setkalo, musí tuto úlohu ihned provést a pozastavit aktuální úlohu, než je dokončen výpočet synovské úlohy. Pokud je podmínka vyhodnocena jako pravdivá, pak je chování direktivy stejné, jako kdyby klauzule **if** nebyla vůbec uvedena, tj. dojde k vytvoření nové úlohy, kterou může rovnou provést dané vlákno, nebo může být úloha odložena do fronty odložených úloh. [14]
- **priority(*expression*)**: Tato klauzule slouží jako nápověda pro prioritu vygenerované úlohy. Ze všech úloh, které jsou připraveny k provedení, se doporučuje nejprve provést úlohy s vyšší prioritou. Pokud není tato klauzule specifikována, je výchozí hodnota priority vygenerované úlohy rovna nule. [14]
- **final(*expression*)**: Pokud je výraz *expression* vyhodnocen jako pravdivý, je daná úloha označena jako finální a všechny úlohy, které jsou vygenerovány v rámci této finální úlohy, musí být provedeny okamžitě daným vláknem a nebudou odkládány do fronty úloh. [14]

Pro direktivu **task** jsou také dostupné klauzule pro deklaraci vlastností proměnných, jako např. **private**, **firstprivate**, **default** a **shared**. Uvnitř direktivy **task** je navíc možné použít direktivu **taskwait**, která vytváří synchronizační bod, na kterém musí vlákno provádějící aktuální úlohu čekat na dokončení výpočtu všech synovských úloh. [14]

3.2.6 Synchronizační prostředky

Stejně jako v jazyce C++ je i v OpenMP často nezbytná synchronizace vláken k zajištění správného pořadí přístupů ke sdíleným datům. V OpenMP jsou prostřednictvím direktiv k dispozici například následující synchronizační prostředky:

- **barrier**: Bariéra je synchronizační bod v programu, kde vlákna čekají na sebe navzájem. Žádné vlákno z týmu vláken, na který se tato direktiva vztahuje, nesmí pokračovat za bariéru, dokud všechna vlákna týmu nedosáhnou tohoto synchronizačního bodu. [15]

```
#pragma omp atomic read
local_var = shared_var;

#pragma omp atomic write
shared_var = new_value;

#pragma omp atomic update
shared_var *= 2;

#pragma omp atomic capture
{
    old_value = shared_var;
    shared_var += 42;
}

#pragma omp atomic capture
{
    shared_var += 42;
    new_value = shared_var;
}
```

■ **Výpis kódu 3.3** Ukázka použití synchronizační direktivy `atomic`.

- **single:** Tato direktiva určuje, že strukturovaný blok kódu, který bezprostředně následuje za touto direktivou, může být proveden pouze jedním vláknem. Není určeno, které konkrétní vlákno daný blok kódu provede, a zvolené vlákno se může pro různá spuštění programu lišit. Zbylá vlákna týmu čekají na bariéře, dokud vlákno provádějící blok kódu spjatý s direktivou `single` nedokončí svůj výpočet. [15]
- **master:** Tato direktiva je podobná předchozí, avšak v případě této direktivy je zaručeno, že daný blok kódu provede pouze hlavní vlákno. Tato direktiva však má jeden významný rozdíl oproti direktivě `single`, kterým je absence implicitní bariéry, tj. ostatní vlákna týmu tento strukturovaný blok kódu přeskakují a rovnou pokračují v dalším výpočtu. Pokud je potřeba, aby ostatní vlákna čekala, až hlavní vlákno dokončí svůj výpočet, je nutné použít explicitní synchronizační bariéru. [15]
- **critical[*name*]:** Pomocí této direktivy lze zajistit, aby více vláken nemohlo současně aktualizovat stejná sdílená data. Strukturovaný blok kódu, který následuje za touto direktivou, se označuje jako kritická oblast (z angl. *critical region*) nebo kritická sekce (z angl. *critical section*). Pokud není specifikován název kritické sekce *name*, je kritická sekce anonymní. Když vlákno potká při výpočtu kritickou sekci, čeká, dokud žádné jiné vlákno neprovádí kód žádné kritické sekce se stejným názvem, tj. je zajištěno, že kód obsažený v kritických sekcích se stejným názvem provádí ve stejný

čas vždy nejvýše jedno vlákno. Pokud existuje více anonymních kritických sekcí, platí vzájemné vyloučení globálně pro všechny její výskyty. [15]

- **taskwait**: Slouží k synchronizaci aktuální úlohy se všemi jejími synovskými úlohami, viz. 3.2.5.
- **atomic**: Zajišťuje, že daná operace bude provedena atomicky, tj. nepřerušitelně. Tato direktiva podporuje následující čtyři tzv. atomické klauzule (z angl. *atomic-clause*), které specifikují druh provedené operace. Pro čtení je určena atomická klauzule **read**, pro zápis **write**, pro aktualizaci hodnoty **update** a pro uložení hodnoty před či po aktualizaci slouží atomická klauzule **capture**. [14] Ukázku použití všech atomických klauzulí ukazuje výpis 3.3.

Podobně jako C++ podporuje OpenMP různé modely paměťového uspořádání, které lze prostřednictvím klauzulí nastavit v direktivách **atomic**. Dále jsou také k dispozici zámky, pokud je potřeba flexibilnější kontrola nad vzájemným vyloučením oproti kritickým sekcím, jako je např. nastavení maximální délky pokusu o získání zámku. Více o těchto a dalších funkcionalitách dostupných v OpenMP se lze dočíst např. v [14].

[illegible]

Základní pojmy

Tato kapitola vysvětluje základní pojmy z teorie složitosti, lineární algebry a zabývá se pojmem řádké matice.

4.1 Teorie složitosti

U algoritmů se běžně zkoumá jejich časová (někdy také operační) a paměťová (též prostorová) složitost. Časová složitost reprezentuje počet elementárních operací, které algoritmus musí vykonat v závislosti na velikosti vstupních dat. Paměťová složitost reprezentuje počet buněk v paměti, které algoritmus potřebuje k vyřešení instance daného problému. Časová, resp. paměťová, složitost obecně závisí nejen na velikosti vstupních dat, ale i na jejich konkrétních hodnotách. Složitost lze proto uvažovat v nejlepším, průměrném nebo nejhorším případě, zpravidla se však uvažuje nejhorší možný případ pro danou konkrétní velikost vstupu. Určení přesné složitosti je u složitějších algoritmů velmi náročnou a někdy i nemožnou úlohou. Protože je navíc z praktických důvodů důležité chování algoritmu pro velké instance dané úlohy, odhaduje se složitost algoritmů pouze asymptoticky. [17, 18]

► **Definice 4.1** (Asymptotická horní mez [18]). *Nechť $f, g : \mathbb{N} \rightarrow \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je nejvýše řádu $g(n)$, psáno $f(n) = O(g(n))$ právě tehdy, když platí následující podmínka:*

$$\exists c \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : c \cdot g(n) \geq f(n).$$

Asymptotická horní mez tedy vyjadřuje, jak roste složitost algoritmu, když velikost vstupních dat roste nad všechny meze. Značení $f(n) = O(g(n))$ je poněkud nepřesné, protože $O(g(n))$ reprezentuje množinu funkcí, a vhodnější by proto bylo množinové značení $f(n) \in g(n)$. Nicméně v literatuře se běžně používá značení s rovnítkem, proto bude použito i v tomto textu. [17, 18]

Analogicky je definována asymptotická dolní mez. Funkce, která je horní a dolní asymptotickou mezí současně, se pak nazývá asymptotická těsná mez.

► **Definice 4.2** (Asymptotická dolní mez [18]). *Nechť $f, g : \mathbb{N} \rightarrow \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je nejméně řádu $g(n)$, psáno $f(n) = \Omega(g(n))$ právě tehdy, když platí následující podmínka:*

$$\exists c \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : f(n) \geq c \cdot g(n).$$

► **Definice 4.3** (Asymptotická těsná mez [18]). *Nechť $f, g : \mathbb{N} \rightarrow \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je stejného řádu jako $g(n)$, psáno $f(n) = \Theta(g(n))$ právě tehdy, když platí následující podmínka.*

$$\exists c_1, c_2 \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n).$$

Pro potřeby vyjádření, že funkce $f(n)$ je striktně nižšího, resp. vyššího řádu než funkce $g(n)$, slouží striktně větší horní mez, resp. striktně menší dolní mez.

► **Definice 4.4** (Striktně větší horní mez [17]). *Nechť $f, g : \mathbb{N} \rightarrow \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je striktně nižšího řádu než řádu $g(n)$, psáno $f(n) = o(g(n))$ právě tehdy, když platí následující podmínka:*

$$\forall c \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : c \cdot g(n) > f(n).$$

► **Definice 4.5** (Striktně menší dolní mez [17]). *Nechť $f, g : \mathbb{N} \rightarrow \mathbb{R}$ jsou dvě reálné funkce. Funkce $f(n)$ je striktně vyššího řádu než funkce $g(n)$, psáno $f(n) = \omega(g(n))$ právě tehdy, když platí následující podmínka:*

$$\forall c \in \mathbb{R}^+ \wedge \exists n_0 \in \mathbb{N} \wedge \forall n \geq n_0 : f(n) > c \cdot g(n).$$

V případě algoritmů nad dynamickou datovou strukturou, jejíž velikost a obsah se v čase mění, nemusí asymptotická horní mez nejlépe vystihovat chování takového algoritmu v rámci celého životního cyklu této datové struktury. Z toho důvodu se u algoritmů nebo operací, které se nad dynamickou datovou strukturou volají mnohokrát, zavádí amortizovaná složitost. [18]

► **Definice 4.6** (Amortizovaná časová složitost [18]). *Operace A nad dynamickou datovou strukturou má v daném kontextu svého provádění amortizovanou časovou složitost $O^*(f(n))$, pokud posloupnost $k \in \mathbb{N}$ operací A má celkovou časovou složitost $O(k \cdot f(n))$. Parametr $n \in \mathbb{N}_0$ je velikost dynamické datové struktury po provedení této posloupnosti operací.*

Čas, který je potřebný k provedení posloupnosti operací nad dynamickou datovou strukturou, se při amortizované analýze zprůměruje přes všechny provedené operace. Amortizovaná časová složitost tedy umožňuje ukázat, že průměrné náklady na operaci jsou malé, pokud jsou tyto náklady zprůměrovány přes celou posloupnost provedených operací. Jednotlivá operace v této posloupnosti provedených operací však může být velmi drahá, a platí, že pro odhad jednotlivých provedení této operace se stále bere nejhorší případ. [18]

Složitost paralelního algoritmu však nezávisí pouze na velikosti vstupu n , ale také na celkovém počtu procesorů, resp. výpočetních jader. Pro vyjádření složitosti paralelního algoritmu slouží paralelní čas. Pro účely této a následujících metrik bude symbol p označovat celkový počet výpočetních jednotek, tedy např. pro systém se dvěma čtyřjádrovými procesory platí, že $p = 8$. [19]

► **Definice 4.7** (Paralelní čas [19]). *Nechť n je velikost vstupu a p počet výpočetních jednotek paralelního výpočetního systému. Čas požadovaný paralelním algoritmem pro dokončení výpočtu na paralelním výpočetním systému se nazývá paralelní čas a značí se $T_{par}(n, p)$.*

Paralelní čas není jedinou metrikou kvality paralelního algoritmu, neboť záleží také na efektivním využití dostupných výpočetních jednotek. Proto existuje míra, která kromě doby běhu paralelního algoritmu zohledňuje i produktivitu výpočetních jednotek.

► **Definice 4.8** (Paralelní cena [19]). *Celkový počet dostupných cyklů během výpočtu paralelního algoritmu se nazývá paralelní cena, která se značí $C(n, p)$ a je definována následovně:*

$$C(n, p) = p \cdot T_{par}(n, p).$$

Další metrikou pro měření kvality paralelního algoritmu je paralelní zrychlení, které je přirozeně definováno jako poměr času sekvenčního řešení k času paralelního řešení.

► **Definice 4.9** (Paralelní zrychlení [19]). *Nechť $T_{seq}(n)$ je čas nejefektivnějšího sekvenčního algoritmu pro řešení daného problému a $T_{par}(n, p)$ je čas nejefektivnějšího paralelního algoritmu pro řešení stejného problému na systému s p výpočetními jednotkami. Pak paralelní zrychlení $S(n, p)$ je definováno jako*

$$S(n, p) = \frac{T_{seq}(n)}{T_{par}(n, p)}.$$

Pokud platí, že $S(n, p) = \Theta(p)$, nazývá se takové zrychlení lineární. Pokud platí $S(n, p) = \omega(p)$, nazývá se takové zrychlení superlineární.

Může se zdát, že sekvenční algoritmus lze napsat tak, aby emuloval paralelní algoritmus se zpomalením $O(p)$, což by mělo za následek, že superlineární zrychlení nemůže nikdy nastat. Nicméně existují výjimečné situace, ve kterých je superlineární zrychlení možné. Za prvé, v paralelním algoritmu, který hledá řešení problému v kombinatorickém stavovém prostoru, může mít určité jádro štěstí a při paralelním výpočtu objevit řešení dříve, než by to dokázala sekvenční emulace daného paralelního algoritmu. Za druhé, mohou se projevit vlivy hierarchie paměti. Například úspěšná posloupnost přístupů do cache pamětí výpočetních jader procesoru (viz. 2.2) může mít drastický vliv na dobu běhu algoritmu. [19]

Poslední mírou kvality paralelního algoritmu je metrika, která vyjadřuje, jak dobře jsou využity výpočetní jednotky paralelního systému.

► **Definice 4.10** (Paralelní efektivnost [19]). *Nechť $T_{seq}(n)$ je čas neefektivnějšího sekvenčního algoritmu pro řešení daného problému a $C(n, p)$ je paralelní cena řešení stejného problému na systému s p výpočetními jednotkami. Pak paralelní efektivnost $E(n, p)$ je definováno jako*

$$E(n, p) = \frac{T_{seq}(n)}{C(n, p)}.$$

4.2 Lineární algebra

Pro účely analýzy všech formulací násobení matic a formátů pro reprezentaci matic je nejprve nutné zavést několik elementárních pojmů z lineární algebry. Formální zavedení těchto pojmů by rovněž vyžadovalo zavedení dalšího poměrně komplexního aparátu od algebraických struktur až po vektorový prostor a prostor se skalárním součinem. Z těchto důvodů budou některé pojmy z lineární algebry pro účely tohoto textu zjednodušeny, nicméně na tato zjednodušení bude upozorněno a čtenář může formální definice nalézt např. v [20].

► **Definice 4.11** (Vektor [20, 21]). *Vektor dimenze n je uspořádaná n -tice reálných čísel s následujícím značením:*

$$\mathbf{x} = (x_1 \ x_2 \ \cdots \ x_n),$$

kde x_i značí i -tý prvek vektoru. Množina všech n -rozměrných vektorů s reálnými prvky se značí $\mathbb{R}^n$.

► **Definice 4.12** (Součet vektorů [20, 21]). *Nechť $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, pak součet těchto vektorů je následující vektor:*

$$\mathbf{x} + \mathbf{y} = (x_1 + y_1 \ x_2 + y_2 \ \cdots \ x_n + y_n).$$

Protože je součet vektorů definován po složkách a sčítání reálných čísel je komutativní a asociativní, je i sčítání vektorů komutativní a asociativní.

► **Definice 4.13** (Násobení vektoru číslem [20, 21]). *Nechť $a \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^n$. Výsledkem násobení reálného čísla s vektorem je následující vektor:*

$$a \cdot (x_1 \ x_2 \ \cdots \ x_n) = (a \cdot x_1 \ a \cdot x_2 \ \cdots \ a \cdot x_n).$$

Formálně se vektory definují jako prvky vektorového prostoru nad algebraickou strukturou, která se nazývá *těleso*. Množina reálných čísel s obvyklými operacemi sčítání a násobení tvoří těleso, a společně s výše definovanými operacemi sčítání vektorů a násobení vektoru reálným číslem pak tvoří vektorový prostor. Proto pro účely této práce bude dostačující uvažovat vektory s prvky z množiny reálných čísel.

► **Definice 4.14** (Lineární kombinace [20, 21]). *Nechť $\mathbf{x} \in \mathbb{R}^n$ a $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ je soubor vektorů z $\mathbb{R}^n$. Vektor $\mathbf{x}$ je lineární kombinací souboru $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ právě tehdy, když existují čísla $\alpha_1, \dots, \alpha_n \in \mathbb{R}$ taková, že*

$$\mathbf{x} = \sum_{i=1}^n \alpha_i \cdot \mathbf{x}_i.$$

Čísla α_i se nazývají koeficienty lineární kombinace.

► **Definice 4.15** (Skalární součin [20, 21]). *Nechť $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, pak skalární součin těchto vektorů je následující reálné číslo:*

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i.$$

Formálně se skalární součin definuje nad vektorovým prostorem jako zobrazení, které splňuje axiomy prostoru se skalárním součinem. Výše definovaný skalární součin se pak nazývá standardní skalární součin.

► **Definice 4.16** (Matice [20, 21]). *Nechť $m, n \in \mathbb{N}$. Posloupnost $m \cdot n$ čísel uspořádaná do tabulky o m řádcích a n sloupcích se nazývá matice o rozměrech $m \times n$. Matice mají následující značení:*

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix},$$

kde a_{ij} značí prvky matice (označované také jako $\mathbf{A}_{ij}$) a nazývají se ij -té prvky pro $i \in \{1, 2, \dots, m\}$ a $j \in \{1, 2, \dots, n\}$. Číslo i , resp. j , se pak nazývá řádkový, resp. sloupcový, index. Množina všech matic o rozměrech $m \times n$ s prvky z množiny reálných čísel se značí $\mathbb{R}^{m,n}$.

Formálně prvky matice pocházejí z algebraické struktury, která se nazývá okruh, nicméně pro účely této práce je dostačující uvažovat množinu reálných čísel s operacemi sčítání a násobení, které axiomy okruhu splňují.

► **Definice 4.17** (Čtvercová matice [20, 21]). *Matice $\mathbf{A} \in \mathbb{R}^{m,n}$ se nazývá čtvercová, pokud $m = n$.*

► **Definice 4.18** (Řád čtvercové matice [21]). *Nechť $\mathbf{A} \in \mathbb{R}^{n,n}$ je čtvercová matice. Číslo n se pak nazývá řád čtvercové matice.*

► **Definice 4.19** (Rovnost matic [20, 21]). *Dvě matice $\mathbf{A}$ a $\mathbf{B}$ se rovnají, pokud jsou stejného rozměru a všechny prvky jedné matice se rovnají odpovídajícím prvkům matice druhé.*

► **Definice 4.20** (Řádkový a sloupcový vektor matice [20, 21]). *Nechť $\mathbf{A}$ je matice z množiny $\mathbb{R}^{m,n}$. $\mathbf{A}_{*j} \in \mathbb{R}^{m,1}$ pro $j \in \{1, 2, \dots, n\}$ značí j -tý sloupcový vektor (nebo zkráceně sloupec) matice $\mathbf{A}$:*

$$\mathbf{A}_{*j} = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix}.$$

$\mathbf{A}_{i*} \in \mathbb{R}^{1,n}$ pro $i \in \{1, 2, \dots, m\}$ značí i -tý řádkový vektor (nebo zkráceně řádek) matice $\mathbf{A}$:

$$\mathbf{A}_{i*} = (a_{i1} \ a_{i2} \ \cdots \ a_{in}).$$

► **Definice 4.21** (Součet matic [20, 21]). *Nechť $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m,n}$. Součtem matic $\mathbf{A}$ a $\mathbf{B}$ je následující matice:*

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{pmatrix}.$$

Podobně jako u součtu vektorů i pro součet matic platí, že z důvodu jeho definice po složkách a komutativity a asociativity sčítání reálných čísel je i součet matic komutativní a asociativní.

► **Definice 4.22** (Diagonála matice [22, 23]). *Nechť $\mathbf{A}$ je matice z množiny $\mathbb{R}^{m,n}$. k -tá diagonála matice $\mathbf{A}$ pro $k \in \{1-m, 2-m, \dots, -1, 0, 1, \dots, n-2, n-1\}$ je následující vektor:*

$$(a_{i_1 j_1} \ a_{i_2 j_2} \ \cdots \ a_{i_l j_l}),$$

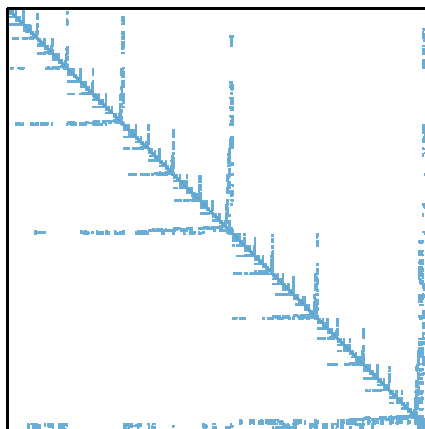
kde $i_1 < i_2 < \dots < i_l$, $j_1 < j_2 < \dots < j_l$, $a_{i_t j_t}$ označuje prvky matice $\mathbf{A}$ a $j_t - i_t = k$ pro $t = 1, 2, \dots, l$.

0-tá diagonála se nazývá hlavní diagonála matice $\mathbf{A}$, k -tá diagonála pro $k < 0$ subdiagonála a k -tá diagonála pro $k > 0$ superdiagonála. Každou z těchto typů je také možné zkráceně označit jako diagonálu. Číslo k se také nazývá offset či index diagonály.

► **Definice 4.23** (Transpozice matice [20, 21]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$. Transpozicí matice $\mathbf{A}$ je matice $\mathbf{A}^T \in \mathbb{R}^{n,m}$, pro kterou platí $(\mathbf{A}^T)_{ij} = \mathbf{A}_{ji}$.*

► **Definice 4.24** (Symetrická matice [20]). *Nechť $\mathbf{A}$ je čtvercová matice řádu n . Matice $\mathbf{A}$ je symetrická, pokud platí $\mathbf{A}^T = \mathbf{A}$.*

Formáty pro ukládání řídkých matic, jako jsou Rutherford–Boeing nebo Matrix Market, využívají symetrie matic tím, že ukládají pouze prvky z jejich dolní, resp. horní trojúhelníkové části, viz. 6.2.



■ **Obrázek 4.1** Řídká matice Chen/pkustk13 pocházející z kolekce SuiteSparse Matrix Collection. [24] Vizualizace vygenerována pomocí nástroje *marc*. [25]

4.3 Řídké matice

Pro účely analýzy výhod a nevýhod všech formulací maticového násobení v kontextu problému násobení řídkých matic je nejprve nutné definovat pojem řídké matice. Problémem je, že pro řídkou matici neexistuje jednotná definice. Jedna z definic např. označuje matici za řídkou, pokud obsahuje mnoho nulových prvků. [26, 27, 28] Další zdroj vztahuje řídkost matice ke způsobu její reprezentace. Konkrétně definuje matici $\mathbf{A} \in \mathbb{R}^{m,n}$ jako hustou, pokud je reprezentována pomocí pole o m řádcích a n sloupcích s $m \cdot n$ prvky. Řídkou matici pak definuje jako matici uloženou v datové struktuře, která využívá řídkosti matice tím, že neukládá její nulové prvky. [23] Další zdroj uvádí, že matice je řídká, pokud je pro její reprezentaci i při výpočtech s touto maticí výhodné využít její nulové prvky. Hustou matici pak definuje jako matici, pro kterou se s žádným jejím nulovým prvkem nevyplatí zvláštní zacházení. [29] Jeden článek pak využívá kontrastu počtu nulových a nenulových prvků tím, že řídkou matici definuje jako matici, ve které je počet nenulových prvků zanedbatelný ve srovnání s počtem nulových prvků. [30]

Je zřejmé, že všechny tyto definice jsou poněkud obecné a nekonkrétní. Formální definici přináší až článek [31], který se zabývá násobením řídké matice a hustého vektoru na grafických kartách. V tomto článku je matice $\mathbf{A} \in \mathbb{R}^{n,n}$ označena jako řídká, pokud má $O(n)$ nenulových prvků. Nicméně tato definice je trochu omezující v tom, že uvažuje pouze čtvercové matice řádu n . Proto pro účely tohoto textu bude tato definice mírně modifikována tak, aby zahrnovala i obecné matice.

► **Definice 4.25** (Řídká matice [31]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$. Matice $\mathbf{A}$ je řídká, pokud počet jejích nenulových prvků je $O(m + n)$.*

Tato definice je výhodná nejen proto, že zahrnuje i matice obecných rozměrů, ale dokonce pro čtvercové matice řádu n přesně odpovídá definici pu-

blikované ve článku [31], neboť $O(n + n) = O(2n)$ se z definice asymptotické horní meze redukuje na $O(n)$.

Vizualizace řídké matice z kolekce *SuiteSparse Matrix Collection* s vyznačením nenulových prvků je k vidění na obrázku 4.1.

Pro analýzu formulací a algoritmů násobení matic je někdy výhodné psát počet nenulových prvků matice $\mathbf{A} \in \mathbb{R}^{m,n}$ ve zkrácené podobě. K tomuto účelu slouží následující funkce.

► **Definice 4.26** (Počet nenulových prvků matice [32]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$ a nnz je funkce $\mathbb{R}^{m,n} \rightarrow \mathbb{N}_0$. Funkční hodnotou $nnz(\mathbf{A})$ je počet nenulových prvků matice $\mathbf{A}$.*

Díky této definici je pak možné pro $\mathbf{A} \in \mathbb{R}^{m,n}$ místo počtu nenulových prvků matice $\mathbf{A}$ psát jednoduše $nnz(\mathbf{A})$.

Další častou využívanou vlastností řídkých matic je jejich hustota, která vyjadřuje poměr nenulových prvků k celkovému počtu prvků matice.

► **Definice 4.27** (Hustota matice [21]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$. Hustota matice $\mathbf{A}$ je podíl*

$$\frac{nnz(\mathbf{A})}{m \cdot n}.$$

Formulace násobení matic

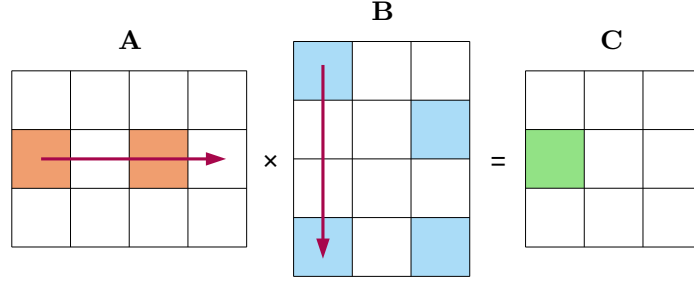
Tato kapitola se zabývá zavedením všech čtyř formulací maticového násobení, jejich popisem, důkazy vzájemné ekvivalence a analýzou těchto formulací pro účely paralelního násobení řídkých matic.

Protože matici je možné procházet přes sloupcové nebo řádkové vektory, existují celkem čtyři různé formulace pro součin matic. Nejprve bude jako základní definice uvedena klasická formulace násobení matic, a u ostatních formulací bude proveden důkaz o jejich ekvivalenci se základní definicí. Je důležité poznamenat, že násobení matic je asociativní operace, ale obecně se nejedná o operaci komutativní. [20, 21]

Každá formulace bude zkoumána především z hlediska vhodnosti použití pro paralelní násobení řídkých matic. Konkrétně budou popsány její výhody, nevýhody, možnosti paralelizace, paměťová náročnost vzhledem k cache paměti a míra znovupoužití dat, která je ve článku [33] definována jako poměr počtu operací násobení a akumulace mezivýsledku (z angl. *multiply-accumulate*, zkr. MAC) k počtu přečtených a zapsaných dat. Pro účely zjednodušení odhadů míry znovupoužití dat a nároků na cache paměť budou uvažována stejně jako ve článku [33] následující zjednodušení:

- Matice $\mathbf{A}$, $\mathbf{B}$ a $\mathbf{C}$ jsou čtvercové řádu n .
- Obě vstupní matice $\mathbf{A}$ a $\mathbf{B}$ mají stejný počet nenulových prvků, tj. platí $nnz(\mathbf{A}) = nnz(\mathbf{B})$.
- Všechny matice mají rovnoměrné rozdělení nenulových prvků, tj. např. pro matici $\mathbf{A}$ platí, že počet nenulových prvků v každém jejím řádku či sloupci je $nnz(\mathbf{A})/n$.

Nicméně tato zjednodušení budou uvažována pouze pro účely odhadu míry znovupoužití dat a paměťové náročnosti, a všechny ostatní úvahy budou platné i bez těchto zjednodušení. Míra znovupoužití dat bude označována jako DR.



■ **Obrázek 5.1** Vizualizace RbC násobení. Překresleno z [34].

(z angl. *data reuse*) a nároky na cache paměť budou označeny jako CMR (z angl. *cache memory requirements*).

5.1 Row-by-Column násobení matic

► **Definice 5.1** (Row-by-Column násobení matic [1, 33]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$. Pak součinem matic, který je založený na součinu řádkového vektoru matice $\mathbf{A}$ a sloupcového vektoru matice $\mathbf{B}$, je matice $\mathbf{C} \in \mathbb{R}^{m,p}$, pro jejíž prvky $\mathbf{C}_{ij}$, kde $i \in \{1, 2, \dots, m\}$ a $j \in \{1, 2, \dots, p\}$, platí:*

$$\mathbf{C}_{ij} = \sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{kj}.$$

*Tento součin se nazývá Row-by-Column (zkr. RbC), případně také vnitřní součin (z angl. *inner product*).*

Z definice tedy plyne, že jeden výpočetní krok RbC násobení je skalární součin řádkového vektoru matice $\mathbf{A}$ a sloupcového vektoru matice $\mathbf{B}$, přičemž výstup jednoho kroku je právě jeden prvek výsledné matice $\mathbf{C}$. Vizualizace RbC násobení je k vidění na obrázku 5.1, pseudokód tohoto násobení pak ukazuje algoritmus 5.1.

Pravděpodobnost, že výstup jednoho kroku RbC násobení bude nenulový prvek, je $nnz(\mathbf{C})/n^2$, proto pro znovupoužití dat, jak uvádí článek [33], platí:

$$DR_{RbC} = O\left(\frac{nnz(\mathbf{C})/n^2}{nnz(\mathbf{A})/n}\right) = O\left(\frac{nnz(\mathbf{C})}{nnz(\mathbf{A})} \cdot \frac{1}{n}\right).$$

Vzhledem k tomu, že rozměr n může u řídkých matic dosahovat stovek tisíc až desítek miliónů, je pro velké matice znovupoužití dat u RbC násobení blízké nule. Výhodou však je, že nároky na cache paměť této formulace jsou také velmi nízké, konkrétně $CMR_{RbC} = O(nnz(\mathbf{A})/n)$, přičemž podíl $nnz(\mathbf{A})/n$ bývá u řídkých matic v řádu desítek až stovek prvků. Druhou výhodou je, že tato formulace nabízí přímočarou paralelizaci, neboť je možné rozdělit výpočet skalárních součinů mezi jednotlivá vlákna a prvky výstupní matice rozdělovat po cache blocích, aby nedocházelo k falešnému sdílení (viz. 2.6). [1, 33, 34]

Vstup : Matice $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$
Výstup: Matice $\mathbf{C} \in \mathbb{R}^{m,p}$

```

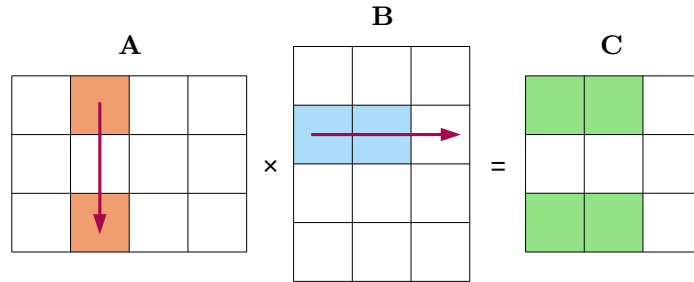
1 Alokuj matici  $\mathbf{C}$ .
2 Nastav prvky matice  $\mathbf{C}$  na nuly.
3 for  $i \leftarrow 1$  to  $m$  do
4   for  $j \leftarrow 1$  to  $p$  do
5     for  $k \leftarrow 1$  to  $n$  do
6        $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} + \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$ 
7     end
8   end
9 end
10 return  $\mathbf{C}$ 
```

Algoritmus 5.1: Pseudokód RbC násobení matic. [34]

Nicméně tato formulace má tři zásadní nevýhody, které ji činí nevhodnou volbou pro paralelní násobení řádkových matic:

1. Obě vstupní matice musí být reprezentovány v různých formátech, matice $\mathbf{A}$ v řádkovém formátu a matice $\mathbf{B}$ ve sloupcovém formátu. [33, 34]
2. RbC násobení je pro husté matice poměrně efektivní, avšak vzhledem k tomu, že RbC násobení v jednom kroku vypočítá jeden prvek výstupní matice, je tato formulace velmi nevhodná pro násobení řídkých matic, neboť vede ke značnému množství zbytečných výpočtů. U skalárního součinu je totiž nezbytné provést iteraci všech prvků řádkového vektoru matice $\mathbf{A}$ souběžně s prvky sloupcového vektoru matice $\mathbf{B}$, a to i v případě, že existuje jen málo reálných průniků, tj. případů, kdy jsou oba prvky z obou vektorů nenulové. Např. na obrázku 5.1 je skalární součin druhého řádkového vektoru matice $\mathbf{A}$ a třetího sloupcového vektoru matice $\mathbf{B}$ nulový, a to přesto, že oba vektory obsahují nenulové prvky, neboť u žádné dvojice nenulových prvků se neshoduje index společné dimenze obou vstupních matic. [33, 34]
3. Pokud jsou matice $\mathbf{A}$ a $\mathbf{B}$ řídké, pak RbC násobení v každém kroku počítá skalární součin dvou řídkých vektorů. Tento součin může být velmi neefektivní, neboť MAC operace vyžaduje shodu indexů nenulových prvků obou řídkých vektorů. Například na obrázku 5.1 skalární součin druhého řádkového vektoru matice $\mathbf{A}$ a prvního sloupcového vektoru matice $\mathbf{B}$ vyžaduje dvě operace pro vyhledání shody indexů z řádkového vektoru matice $\mathbf{A}$, avšak provede se pouze jedna MAC operace. [33, 34]

Z těchto důvodů není RbC násobení vhodné pro paralelní násobení řídkých matic a je proto druhou nejméně studovanou formulací paralelního součinu řídkých matic. [1]



■ **Obrázek 5.2** Vizualizace CbR násobení. Překresleno z [34].

5.2 Column-by-Row násobení matic

► **Definice 5.2** (Column-by-Row násobení matic [1, 33]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$. Pak součinem matic, který je založený na součinu sloupcového vektoru matice $\mathbf{A}$ a řádkového vektoru matice $\mathbf{B}$, je matice $\mathbf{C} \in \mathbb{R}^{m,p}$, která se získá jako následující součet matic:*

$$\mathbf{C} = \sum_{k=1}^n \mathbf{A}_{*k} \mathbf{B}_{k*}.$$

Tento součin se nazývá Column-by-Row (zkr. CbR), případně také vnější součin (z angl. outer product).

Z definice tedy plyne, že jeden výpočetní krok CbR násobení je součin sloupcového vektoru matice $\mathbf{A}$ a řádkového vektoru matice $\mathbf{B}$, přičemž výstup jednoho kroku je dílčí sčítanec celé výstupní matice $\mathbf{C}$. Vizualizace CbR násobení je k vidění na obrázku 5.2, pseudokód tohoto maticového násobení pak ukazuje algoritmus 5.2.

► **Tvrzení 5.3.** *Column-by-Row a Row-by-Column formulace násobení matic jsou ekvivalentní.*

Důkaz. Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$ a $\mathbf{C} \in \mathbb{R}^{m,p}$. Dále nechť $i \in \{1, 2, \dots, m\}$ a $j \in \{1, 2, \dots, p\}$. Pak pro ij -tý prvek výsledku CbR násobení platí:

$$C_{ij} = \left(\sum_{k=1}^n \mathbf{A}_{*k} \mathbf{B}_{k*} \right)_{ij} = \sum_{k=1}^n (\mathbf{A}_{*k} \mathbf{B}_{k*})_{ij} = \sum_{k=1}^n (\mathbf{A}_{*k})_i (\mathbf{B}_{k*})_j = \sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{kj}.$$

□

Řádek a sloupec každé ze vstupních matic má $nnz(\mathbf{A})/n$ nenulových prvků, proto každý krok CbR násobení provede $(nnz(\mathbf{A})/n)^2$ MAC operací. Znovupoužití dat této formulace tedy je $DR_{CbR} = O(nnz(\mathbf{A})/n)$. Tato formulace má tedy vynikající znovupoužití dat, dokonce nejlepší ze všech čtyř formulací. Podíl $nnz(\mathbf{A})/n$ totiž u řídkých matic používaných v praxi bývá několik desítek

Vstup : Matice $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$
Výstup: Matice $\mathbf{C} \in \mathbb{R}^{m,p}$

```

1 Alokuj matici  $\mathbf{C}$ .
2 Nastav prvky matice  $\mathbf{C}$  na nuly.
3 for  $i \leftarrow 1$  to  $n$  do
4   for  $j \leftarrow 1$  to  $m$  do
5     for  $k \leftarrow 1$  to  $p$  do
6        $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} + \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$ 
7     end
8   end
9 end
10 return  $\mathbf{C}$ 
```

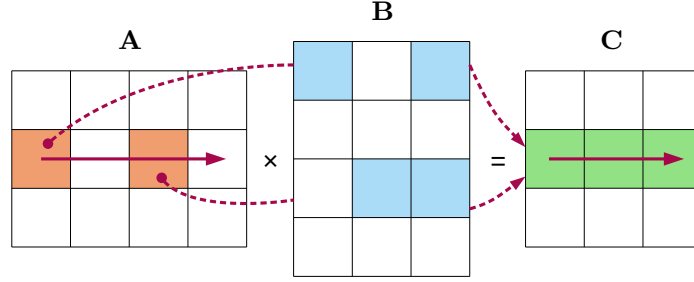
Algoritmus 5.2: Pseudokód CbR násobení matic. [34]

až stovek prvků, proto CbR násobení data zapsaná do paměti či přečtená z paměti znovu využije desetkrát až stokrát. Nároky na cache paměť jsou však také vysoké, konkrétně $CMR_{CbR} = O(nnz(\mathbf{C}) + nnz(\mathbf{A})/n)$, tj. nejvyšší nároky ze všech čtyř formulací. Paralelizace CbR násobení je také problematická, protože ačkoliv lze výpočet dílčích součtů výsledné matice $\mathbf{C}$ rozdělit mezi jednotlivá vlákna, je potřeba zajistit exkluzivní přístup ke všem buňkám výstupní matice, aby nedocházelo k časově závislým chybám. [1, 33, 34]

CbR násobení řeší dva hlavní problémy RbC násobení. Za prvé, CbR násobení v jednom kroku vypočítává nenulové položky výstupní matice, neboť dílčí sčítanec výsledné matice je nulový pouze tehdy, když je sloupcový nebo řádkový vektor nulový, což je málo pravděpodobné. Za druhé, CbR formulace při násobení řídkých vstupních vektorů nemusí provádět operace pro vyhledávání a porovnávání indexů. [33, 34].

Nicméně CbR formulace má tři následující klíčové nevýhody:

1. Stejně jako u RbC násobení je i v tomto případě potřeba, aby vstupní matice byly reprezentovány v odlišných formátech, a to jedna ve sloupcovém a druhá v řádkovém formátu. [33, 34]
2. Aby mohla různá vlákna počítat dílčí sčítance výstupní matice, je potřeba přístup ke každé sdílené buňce matice $\mathbf{C}$ ošetřit synchronizačními prostředky, což může významně snižovat výkon celého algoritmu. [33, 34]
3. Při výpočtech dílčích mezivýsledků matice $\mathbf{C}$ je možné se nadměrné synchronizační režii vláken vyhnout tím, že výsledek dílčího součtu bude každé vlákno zapisovat do svého vlastního bufferu. Po té, co všechna vlákna vypočítají svůj částečný výsledek, dojde k sečtení všech dílčích mezivýsledků do výsledné matice $\mathbf{C}$. Nicméně tento přístup je nevhodný nejen kvůli razantnímu zvýšení paměťové náročnosti, ale navíc je problematický, pokud dílčí součty matice $\mathbf{C}$ mají být řídké, neboť sčítání mezivýsledných matic v řídkém formátu je velmi komplexní operace. [34]



■ **Obrázek 5.3** Vizualizace RbR násobení. Překresleno z [34].

CbR násobení tedy sice řeší některá slabá místa RbC formulace, nicméně vzhledem k výše uvedeným problémům ani CbR formulace není ideální volbou pro paralelní násobení řídkých matic.

5.3 Row-by-Row násobení matic

► **Definice 5.4** (Row-by-Row násobení matic [1, 33]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$ a dále $\mathbf{B} \in \mathbb{R}^{n,p}$. Pak součinem matic, který je založený na součinu řádkového vektoru matice $\mathbf{A}$ a všech řádkových vektorů matice $\mathbf{B}$, je matice $\mathbf{C} \in \mathbb{R}^{m,p}$, pro jejíž i -tý řádek, kde $i \in \{1, 2, \dots, m\}$, platí:*

$$\mathbf{C}_{i*} = \sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{k*}.$$

Tento součin se nazývá Row-by-Row (zkr. RbR), případně také Gustavsonův algoritmus.

Z definice tedy plyne, že v jednom výpočetním kroku RbR násobení je vypočítán celý jeden řádek výsledné matice $\mathbf{C}$, přičemž tento výstupní řádek je lineární kombinací řádkových vektorů matice $\mathbf{B}$ a koeficienty této lineární kombinace jsou prvky daného řádkového vektoru matice $\mathbf{A}$. Vizualizace RbR násobení je k vidění na obrázku 5.3, pseudokód tohoto násobení pak ukazuje algoritmus 5.3.

► **Tvrzení 5.5.** *Row-by-Row a Row-by-Column formulace násobení matic jsou ekvivalentní.*

Důkaz. Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$ a $\mathbf{C} \in \mathbb{R}^{m,p}$. Dále nechť $i \in \{1, 2, \dots, m\}$ a $j \in \{1, 2, \dots, p\}$. Pak pro j -tý prvek jednoho kroku RbR násobení, tj. pro j -tý prvek i -tého řádku matice $\mathbf{C}$, platí:

$$(\mathbf{C}_{i*})_j = \left(\sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{k*} \right)_j = \sum_{k=1}^n \mathbf{A}_{ik} (\mathbf{B}_{k*})_j = \sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{kj}.$$

□

Vstup : Matice $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$
Výstup: Matice $\mathbf{C} \in \mathbb{R}^{m,p}$

```

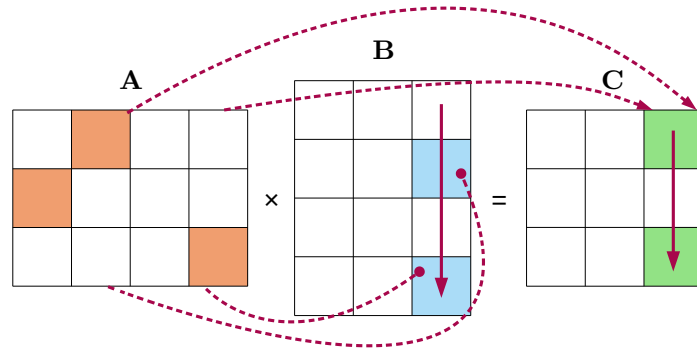
1 Alokuj matici  $\mathbf{C}$ .
2 Nastav prvky matice  $\mathbf{C}$  na nuly.
3 for  $i \leftarrow 1$  to  $m$  do
4   for  $j \leftarrow 1$  to  $n$  do
5     for  $k \leftarrow 1$  to  $p$  do
6        $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} + \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$ 
7     end
8   end
9 end
10 return  $\mathbf{C}$ 
```

Algoritmus 5.3: Pseudokód RbR násobení matic. [34]

V RbR násobení se tedy pro každý nenulový prvek řádkového vektoru matice $\mathbf{A}$ provede součin daného prvku a odpovídajícího řádkového vektoru matice $\mathbf{B}$. Protože matice $\mathbf{B}$ má v jednom řádku $nnz(\mathbf{B})/n$ nenulových prvků, počet vykonaných MAC operací při násobení vektoru číslem je $nnz(\mathbf{B})/n$. Z toho plyne, že při uvažovaných zjednodušeních je znovupoužití dat $DR_{RbR} = O(1)$, tj. nižší než u CbR násobení, ale vyšší než v případě RbC násobení. Nároky na cache paměť jsou $CMR_{RbR} = O(\frac{nnz(\mathbf{B})}{n} + \frac{nnz(\mathbf{C})}{n})$, kde první sčítanec reprezentuje nenulové prvky řádku matice $\mathbf{B}$ a druhý sčítanec řádek matice $\mathbf{C}$. Stejně jako v případě znovupoužití dat je tedy RbR násobení i z hlediska nároků na cache paměť kompromisem mezi oběma extrémy RbC a CbR násobení. Paralelizace je v tomto případě poměrně přímočará, neboť je možné mezi vlákna rozdělit výpočet jednotlivých řádků výstupní matice $\mathbf{C}$. [1, 33, 34]

RbR násobení má tedy následující klíčové výhody:

1. RbR formulace přistupuje ke všem třem maticím v řádkovém pořadí, což umožňuje využít stejný řádkový formát pro reprezentaci jak vstupních, tak výstupních matic. [33, 34]
2. Dílčím krokem RbR formulace je součin čísla a vektoru. Pokud jsou vstupní matice řídké, pak tato formulace vypočítává pouze nenulové prvky výstupní matice, pokud neuvažujeme situaci, kdy se přeskálované prvky sečtou na nulu. Navíc při násobení čísla a vektoru není potřeba provádět žádné operace vyhledání a porovnání indexů jako u RbC násobení. [33, 34]
3. Pokud bude mezi jednotlivá vlákna rozdělen výpočet dílčích řádků výstupní matice $\mathbf{C}$, pak tato formulace nepotřebuje žádnou synchronizaci pro čtení a zápisy do paměti pro výstupní matici. [33]
4. RbR formulace má poměrně nízké nároky na cache paměť, neboť zatímco CbR násobení potřebuje pracovat s celou výstupní maticí, RbR násobení potřebuje do cache uložit pouze jeden řádek výsledné matice $\mathbf{C}$. [33, 34]



■ **Obrázek 5.4** Vizualizace CbC násobení. Inspirováno ilustracemi z [34].

Z těchto důvodů je RbR formulace vhodnou volbou pro paralelní násobení řídkých matic a ze všech čtyř formulací je také tou nejstudovanější. [1]

5.4 Column-by-Column násobení matic

► **Definice 5.6** (Column-by-Column násobení matic [1, 33]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$. Pak součinem matic, který je založený na součinu sloupcového vektoru matice $\mathbf{B}$ a všech sloupcových vektorů matice $\mathbf{A}$, je matice $\mathbf{C} \in \mathbb{R}^{m,p}$, pro jejíž j -tý sloupec, kde $j \in \{1, 2, \dots, p\}$, platí:*

$$\mathbf{C}_{*j} = \sum_{k=1}^n \mathbf{A}_{*k} \mathbf{B}_{kj}.$$

Tento součin se nazývá Column-by-Column (zkr. CbC).

Z definice tedy plyne, že v jednom výpočetním kroku CbC násobení je vypočítán celý jeden sloupec výsledné matice $\mathbf{C}$, přičemž tento výstupní sloupec je lineární kombinací sloupcových vektorů matice $\mathbf{A}$ a koeficienty této lineární kombinace jsou prvky daného sloupcového vektoru matice $\mathbf{B}$. Vizualizace CbC násobení je k vidění na obrázku 5.4, pseudokód tohoto násobení pak ukazuje algoritmus 5.4.

► **Tvrzení 5.7.** *Column-by-Column a Row-by-Column formulace násobení matic jsou ekvivalentní.*

Důkaz. Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$ a $\mathbf{C} \in \mathbb{R}^{m,p}$. Dále $i \in \{1, 2, \dots, m\}$ a dále $j \in \{1, 2, \dots, p\}$. Pak pro i -tý prvek jednoho kroku CbC násobení, tj. pro i -tý prvek j -tého sloupce matice $\mathbf{C}$, platí:

$$(\mathbf{C}_{*j})_i = \left(\sum_{k=1}^n \mathbf{A}_{*k} \mathbf{B}_{kj} \right)_i = \sum_{k=1}^n (\mathbf{A}_{*k})_i \mathbf{B}_{kj} = \sum_{k=1}^n \mathbf{A}_{ik} \mathbf{B}_{kj}.$$

□

Vstup : Matice $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$
Výstup: Matice $\mathbf{C} \in \mathbb{R}^{m,p}$
 1 Alokuj matici $\mathbf{C}$.
 2 Nastav prvky matice $\mathbf{C}$ na nuly.
 3 **for** $i \leftarrow 1$ **to** p **do**
 4 **for** $j \leftarrow 1$ **to** n **do**
 5 **for** $k \leftarrow 1$ **to** m **do**
 6 $\mathbf{C}_{ij} \leftarrow \mathbf{C}_{ij} + \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$
 7 **end**
 8 **end**
 9 **end**
 10 **return** $\mathbf{C}$

Algoritmus 5.4: Pseudokód CbC násobení matic.

Tato formulace je velmi podobná RbR násobení a proto pro ni platí také stejné úvahy jako v případě RbR formulace, včetně nároků na cache paměť a míru znovupoužití dat. Vzhledem k podobnosti s RbR násobením a stejným vlastnostem je CbC násobení tou nejméně studovanou formulací pro paralelní násobení řídkých matic. [1]

5.5 Násobení řídkých matic

Na závěr kapitoly o násobení matic je vhodné ještě definovat samotné násobení řídkých matic.

► **Definice 5.8** (Násobení řídkých matic [5]). *Nechť $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$ a $\mathbf{C} \in \mathbb{R}^{m,p}$. Součin $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ se nazývá násobení řídkých matic (z angl. *sparse general matrix-matrix multiplication*, zkr. *SpGEMM*), pokud $\mathbf{A}$, $\mathbf{B}$ i $\mathbf{C}$ jsou řídké matice.*

Formáty pro reprezentaci a ukládání řídkých matic

Tato kapitola se zabývá popisem a analýzou formátů pro reprezentaci řídkých matic včetně souborových formátů pro jejich ukládání a sdílení.

Dimenze řídkých matic může dosahovat i několika jednotek nebo dokonce desítek miliónů, proto je velmi neefektivní a v některých případech dokonce nemožné reprezentovat takovou matici v paměti počítače se všemi jejími prvky. Vzhledem k tomu, že většina prvků v řídkých maticích je nulová, je pro efektivní výpočty a algoritmy nezbytné tyto matice v paměti reprezentovat pouze prostřednictvím jejich nenulových prvků. Tyto způsoby reprezentace se označují jako úložná schémata, příp. úložné formáty (z angl. *storage schemes*, příp. *storage formats*). Nicméně toto označení může být trochu zavádějící, protože tato schémata slouží k obecné reprezentaci řídké matice v paměti počítače. Ke skutečnému uložení řídké matice do souboru slouží souborové formáty, které pak mohou tuto řídkou matici v souboru reprezentovat pomocí některého existujícího úložného schématu. Například souborový formát *Matrix Market* jako jeden ze způsobů reprezentace řídké matice využívá souřadnicové schéma COO.

Pro účely popisu a analýzy všech formátů bude uvažována matice obecných rozměrů $\mathbf{A} \in \mathbb{R}^{m,n}$.

6.1 Úložná schémata řídkých matic

Pro efektivní práci s řídkými maticemi vznikla řada formátů pro jejich reprezentaci v paměti, přičemž každý formát disponuje odlišnými vlastnostmi a je vhodný pro určité operace s řídkými maticemi. V této kapitole budou popsány nejznámější a nejdůležitější formáty včetně jejich vlastností, paměťové náročnosti a vhodnosti použití pro algoritmy násobení řídkých matic. Paměťová náročnost daného schématu bude označována jako S (z angl. *storage*).

	1	2	3	4	5	6	
	0	6	0	0	0	0	Řádkové indexy
1	0	6	0	0	0	0	3 3 2 5 5 1 3 5
2	0	3	0	0	0	0	Sloupcové indexy
3	2	1	0	7	0	0	2 1 2 5 6 2 4 4
4	0	0	0	0	0	0	Data
5	0	0	0	8	4	5	1 2 3 4 5 6 7 8

■ **Obrázek 6.1** Příklad reprezentace řídké matice ve formátu COO.

6.1.1 Coordinate Format

Nejjednodušším a nejvíce intuitivním způsobem reprezentace řídké matice je souřadnicový formát (z angl. *coordinate format*, zkr. COO). Tento formát reprezentuje nenulové prvky matice jako množinu trojic (i, j, a_{ij}) , kde i značí řádkový index a j sloupcový index prvku $a_{ij} \neq 0$. Pro tuto reprezentaci jsou zapotřebí tři pole, dvě s celočíselnými prvky pro ukládání indexů a jedno s reálnými prvky pro reprezentaci hodnot, přičemž každé pole má právě $nnz(\mathbf{A})$ prvků. [26, 29]

Ačkoliv je pro potřeby daného algoritmu nebo operace nad řídkými maticemi obvyklé, že jsou jednotlivé trojice uspořádány podle svých řádkových a sloupcových indexů, je potřeba zmínit, že takové uspořádání tento formát nezaručuje a trojice jsou obecně neuspořádané. [26, 29, 31, 35] Příklad reprezentace matice ve formátu COO ukazuje obrázek 6.1.

Pokud bude každý řádkový, resp. sloupcový index vyžadovat S_{idx} bajtů paměti a S_{val} bude označovat velikost reprezentace nenulového prvku matice, pak paměťová náročnost reprezentace matice $\mathbf{A}$ ve formátu COO, jak bylo popsáno ve článku [36], je následující:

$$S_{COO} = nnz(\mathbf{A}) \cdot (2 \cdot S_{idx} + S_{val}).$$

Přestože je tento formát intuitivní a velmi jednoduchý na konstrukci, má dvě zásadní nevýhody:

1. Pokud řídká matice obsahuje velké množství nenulových prvků v určitých svých řádcích, resp. sloupcích, pak uložení stejného indexu pro každý nenulový prvek v takovém řádku, resp. sloupci vede k významnému plýtvání pamětí a ke zvýšené režii při přístupu k této paměti. [36]
2. COO formát je velmi nepraktický, pokud potřebujeme k matici přistupovat po jednotlivých řádcích či sloupcích. [29]

Z těchto důvodů se COO schéma pro násobení řídkých matic příliš nepoužívá a obecně je tento formát použitelný jen pro malou množinu operací s říd-

	1	2	3	4	5	6	
							Offsety
1	0	6	0	0	0	0	-2
2	0	3	0	0	0	0	-1
3	2	1	0	7	0	0	0
4	0	0	0	0	0	0	1
5	0	0	0	8	4	5	
							Data
							2
							0
							0
							1
							0
							8
							0
							0
							3
							0
							0
							4
							0
							0
							6
							0
							7
							0
							5
							0

■ **Obrázek 6.2** Příklad reprezentace řídké matice ve formátu DIA.

kými maticemi, jako je např. násobení řídké matice hustým vektorem (angl. *sparse matrix-vector multiplication*, zkr. SpMV). Nicméně i pro násobení řídké matice hustým vektorem je tento formát poměrně problematický, protože většina SpMV algoritmů, které používají schéma COO, rozděluje výpočet mezi vlákna po rovnoměrném počtu nenulových prvků vstupní matice $\mathbf{A}$. K tomu je nezbytná synchronizace nebo atomické operace pro slučování mezivýsledků do výstupního vektoru, což vede k významnému snížení výkonu těchto SpMV algoritmů. [1, 29, 36]

6.1.2 Diagonal Format

Pokud jsou nenulové prvky matice $\mathbf{A}$ soustředěny na menším počtu diagonál, pak se nabízí způsob reprezentace této matice prostřednictvím jejích diagonál s nenulovými prvky. Tento princip reprezentace využívá diagonální formát (z angl. *diagonal format*, zkr. DIA). [35, 36]

DIA formát matici reprezentuje pomocí dvou polí. V poli *data* jsou uloženy samotné diagonály s nenulovými prvky a v poli *offset* jsou uloženy offsety těchto diagonál (viz. definice diagonály 4.22). [35, 36, 37] Ke každé diagonále jsou zleva a zprava doplněny nuly tak, aby takto zarovnaná diagonála začínala v prvním sloupci a končila v posledním sloupci matice. Toto doplnění nul ke každé diagonále má za následek, že každá diagonála matice $\mathbf{A} \in \mathbb{R}^{m,n}$ je v poli *data* uložena pomocí n prvků. Ukázka použití formátu DIA je k vidění na obrázku 6.2.

Pokud N_{dia} bude označovat počet diagonál s alespoň jedním nenulovým prvkem matice $\mathbf{A}$, pak paměťová náročnost DIA formátu, jak bylo popsáno ve článku [36], je následující:

$$S_{DIA} = N_{dia} \cdot n \cdot S_{val} + N_{dia} \cdot S_{idx},$$

kde S_{val} označuje počet bajtů pro reprezentaci nenulového prvku a S_{idx} představuje požadovanou velikost pro uložení jednoho offsetu.

DIA formát díky offsetům diagonál vůbec nepotřebuje uchovávat indexy nenulových prvků a je ideální volbou pro řídké matice, jejichž prvky se primárně nacházejí na menším počtu diagonál. Tento formát je proto využíván např. pro reprezentaci matic, které jsou výsledkem metody konečných diferencí, což je jedna z diskretizačních metod pro numerické řešení diferenciálních rovnic. [37] Nicméně tento formát má dvě zásadní nevýhody:

1. DIA formát je použitelný jen pro velmi malé množství řídkých matic, které mají diagonální strukturu. [36, 37]
2. Pro matice, které mají nenulové prvky rozmístěné na větším množství diagonál, přináší zarovnání všech diagonál nulovými prvky výrazné zvýšení paměťových nároků. Navíc, pokud má matice $\mathbf{A}$ na uložených diagonálách pouze pár nenulových prvků, ukládá DIA formát kromě těchto nenulových prvků i zbytečné nulové prvky, což společně s nulami na zarovnání dále zvyšuje paměťovou náročnost a významně zvyšuje množství zbytečných přístupů do paměti. [36, 37]

Z těchto důvodů se DIA formát pro násobení řídkých matic téměř vůbec nepoužívá, nicméně pro řídké matice, které mají diagonální strukturu, se tento formát někdy používá pro násobení řídké matice hustým vektorem. [1, 37]

6.1.3 ELLPACK

Zatímco DIA formát byl vhodný pro malé množství matic a tím i pro specifické problémové domény, formát ELLPACK (zkr. ELL) je mnohem obecnější. Tento formát reprezentuje matici $\mathbf{A} \in \mathbb{R}^{m,n}$ s nejvýše nnz_{row} nenulovými prvky v každém řádku pomocí dvou dvourozměrných polí *colIdx* a *data*, přičemž každé z těchto polí má právě m řádků a nnz_{row} sloupců. V každém řádku pole *data* jsou uloženy nenulové prvky daného řádku matice $\mathbf{A}$ a tyto prvky jsou zarovnány zprava nulami, pokud je počet nenulových prvků v tomto řádku menší než nnz_{row} . Podobně, v každém řádku pole *colIdx* jsou pak uloženy sloupcové indexy nenulových prvků z daného řádku matice a každý řádek tohoto pole je opět zarovnán zprava nulami, pokud je počet nenulových prvků v tomto řádku menší než nnz_{row} . [31, 36, 37, 35] Ukázka reprezentace řídké matice ve formátu ELL je k vidění na obrázku 6.3.

Nároky na paměť formátu ELL pro reprezentaci matice $\mathbf{A} \in \mathbb{R}^{m,n}$ jsou následující:

$$S_{ELL} = (S_{idx} + S_{val}) \cdot m \cdot nnz_{row},$$

kde S_{idx} označuje potřebnou paměť pro sloupcový index a S_{val} reprezentuje počet bajtů pro jeden nenulový prvek. [36]

	1	2	3	4	5	6	Sloupcové indexy	Data	
1	0	6	0	0	0	0	2	0	0
2	0	3	0	0	0	0	2	0	0
3	2	1	0	7	0	0	1	2	4
4	0	0	0	0	0	0	0	0	0
5	0	0	0	8	4	5	4	5	6

■ **Obrázek 6.3** Příklad reprezentace řídké matice ve formátu ELL.

Protože obě dvourozměrná pole obsahují informace pro jednotlivé řádky matice $\mathbf{A}$, nepotřebuje tento formát vůbec ukládat řádkové indexy. [36] Avšak tento formát má tyto důležité nevýhody:

1. ELL formát významně plýtvá pamětí pro reprezentaci matic, které mají vyšší počet nulových řádků.
2. Tento formát je velice citlivý na parametr nnz_{row} a pokud má matice $\mathbf{A}$ vysoký počet řádků, které mají nenulových prvků výrazně méně než nnz_{row} , pak potřebné zarovnání nulovými prvky v obou polích výrazně plýtvá pamětí a navyšuje zbytečné přístupy do paměti. [36]
3. V extrémním případě, kdy má matice $\mathbf{A}$ jeden řádek plný nenulových prvků, jsou paměťové nároky ELL schématu dokonce vyšší než úplná reprezentace řídké matice $\mathbf{A}$ pomocí dvourozměrného pole včetně jejích nulových prvků. [31]

Je zřejmé, že ELL formát je nejvýhodnější pro matice, které mají nenulové prvky rozmístěné rovnoměrně po všech svých řádcích. ELL formát je tedy podobně jako DIA schéma vhodný pro užší skupinu matic, proto není pro násobení řídkých matic příliš využíván. [1]

6.1.4 Compressed Sparse Row

Jedním z nejužívanějších formátů pro reprezentaci řídkých matic je formát komprimovaných řádků (z angl. *compressed sparse row*, zkr. CSR) nebo také schéma komprimovaného ukládání řádků (z angl. *compressed row storage*, zkr. CRS). Toto schéma optimalizuje souřadnicový formát tím, že pole pro sloupcové indexy a pro data ponechává beze změny, avšak komprimuje údaje v poli řádkových indexů. V tomto poli se neukládá index řádku každého nenulového prvku, ale řádkové ukazatele, které uchovávají pozici, na které začíná první nenulový prvek daného řádku uvnitř zbývajících dvou polí. V poli řádkových

	1	2	3	4	5	6	
							Ukazatele řádků
1	0	6	0	0	0	0	1 2 3 6 6 9
2	0	3	0	0	0	0	
							Sloupcové indexy
3	2	1	0	7	0	0	2 2 1 2 4 4 5 6
4	0	0	0	0	0	0	
							Data
5	0	0	0	8	4	5	6 3 2 1 7 8 4 5

■ **Obrázek 6.4** Příklad reprezentace řídké matice ve formátu CSR.

ukazatelů tedy i -tý záznam pro $i \in \{1 \dots m\}$ matice $\mathbf{A} \in \mathbb{R}^{m,n}$ ukazuje na pozici začátku i -tého řádku v rámci polí pro data a pro sloupcové indexy. Na poslední pozici pole řádkových ukazatelů je zarážka, která ukazuje za poslední prvek zbývajících dvou polí. Velikost pole řádkových ukazatelů pro matici $\mathbf{A}$ je proto $n + 1$ a poslední prvek tohoto pole je vždy pevně nastaven na hodnotu $nnz(\mathbf{A}) + 1$. [1, 26, 31, 36]

Pole sloupcových indexů má tedy $nnz(\mathbf{A})$ prvků a jsou v něm uloženy sloupcové indexy nenulových prvků prvního řádku matice, následované sloupcovými indexy nenulových prvků druhého řádku matice, a tak dále (bez mezer mezi jednotlivými řádky). Obvykle, avšak ne vždy, jsou záznamy každého dílčího řádku uloženy vzestupně podle hodnoty sloupcového indexu. V poli pro data jsou pak uloženy hodnoty odpovídajících nenulových prvků ve stejném pořadí. [26, 31, 36] Ukázka reprezentace řídké matice ve formátu CSR je k vidění na obrázku 6.4.

Paměťová náročnost CSR schématu je následující:

$$S_{CSR} = (m + 1) \cdot S_{idx} + nnz(\mathbf{A}) \cdot (S_{val} + S_{idx}),$$

kde S_{idx} opět představuje počet bajtů pro uložení jednoho celého čísla a S_{val} počet bajtů pro reprezentaci nenulového prvku matice. [36]

CSR formát je jedním z nejpoužívanějších formátů pro reprezentaci řídkých matic. [31, 36, 38] Jeho výhody jsou následující:

1. Tento formát neklade žádné specifické požadavky na strukturu řídké matice, je velmi málo citlivý na rozložení nenulových prvků a na rozdíl od ostatních schémat je proto všestranným formátem, neboť dokáže reprezentovat široké spektrum řídkých matic. [1, 37, 38]
2. Na rozdíl od formátů DIA a ELL umožňuje formát CSR efektivně reprezentovat matice s proměnlivým počtem nenulových prvků na řádek, aniž by kvůli tomuto nerovnoměrnému rozložení nenulových prvků zbytečně plýtvat pamětí. [37]

3. Díky řádkovým ukazatelům nepotřebuje CSR formát podobně jako ELL ukládat žádné řádkové indexy nenulových prvků. [36]
4. CSR formát reprezentuje matici po jednotlivých řádcích a je proto ideální volbou pro algoritmy, které k řídké matici přistupují po řádcích. Navíc je obzvláště výhodný pro RbR formulaci násobení matic, protože umožňuje přístup k řádku v konstantním čase $O(1)$. [1]

Ačkoliv má CSR mnoho výhod, i tento formát má dvě menší slabiny:

1. CSR ukládá v poli řádkových ukazatelů záznam pro každý řádek reprezentované matice. Pokud tato matice obsahuje velké množství nulových řádků, pak se v poli řádkových ukazatelů zbytečně plýtvá pamětí. Nicméně toto plýtvání je výrazně nižší než u ostatních formátů.
2. Pokud má reprezentovaná matice nerovnoměrnou strukturu s výrazně odlišnými počty nenulových prvků na řádek, mohou paralelní algoritmy, které rozdělují výpočet po jednotlivých řádcích, trpět nerovnoměrným rozložením výpočetní zátěže. [36, 37]

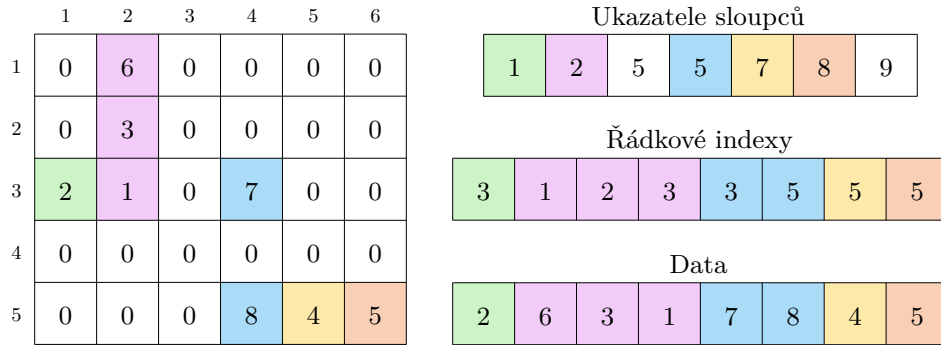
Tyto drobné nevýhody jsou však převáženy dobrými vlastnostmi tohoto formátu a CSR schéma je proto tím nejstudovanějším a nejpoužívanějším formátem pro násobení řídkých matic. [1]

6.1.5 Compressed Sparse Column

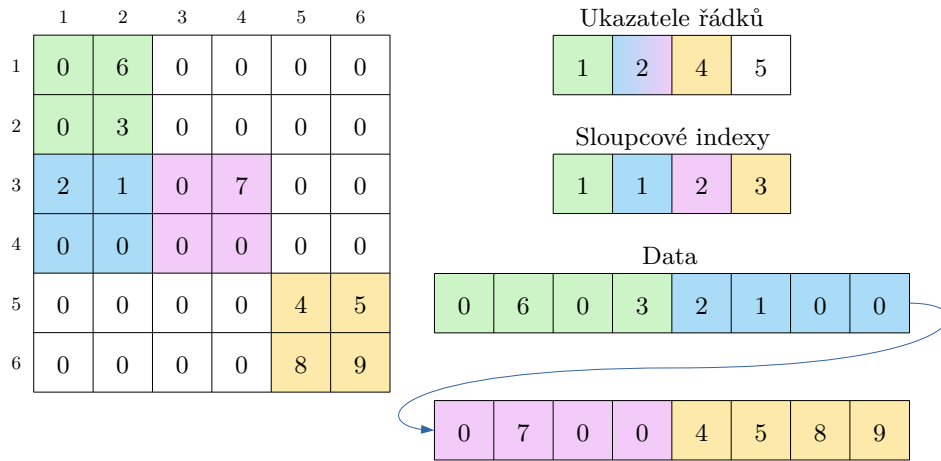
Formát komprimovaných sloupců (z angl. *compressed sparse column*, zkr. CSC) nebo také schéma komprimovaného ukládání sloupců (z angl. *compressed column storage*, zkr. CCS) je definován analogicky k formátu CSR s tím rozdílem, že záznamy ukládá po sloupcích místo po řádcích. V CSC schématu jsou tedy ve srovnání s COO formátem ponechána beze změny pole řádkových indexů a pole dat, a pole sloupcových indexů je nahrazeno polem sloupcových ukazatelů. [26, 31] Ukázka reprezentace řídké matice ve formátu CSC je k vidění na obrázku 6.5.

Vzhledem k tomu, že CSC formát využívá pro reprezentaci řídkých matic stejnou myšlenku jako formát CSR, pouze aplikovanou na sloupce, platí pro CSC tytéž úvahy, které byly uvedeny v kapitole 6.1.4, a to včetně paměťové náročnosti.

Podobně jako je CSR formát ideální volbou pro RbR násobení matic, je CSC formát nejvhodnější pro CbC násobení a lze jej také využít jako reprezentaci matice $\mathbf{B}$ pro RbC násobení matic. Nicméně kvůli své podobnosti s CSR schématem není CSC natolik studovaný, jako je jeho řádková varianta. Hlavním důvodem použití tohoto formátu namísto CSR je to, že některé programovací jazyky, zejména Fortran, tradičně ukládají matice po jednotlivých sloupcích a nikoliv po řádcích. [1, 38]



■ **Obrázek 6.5** Příklad reprezentace řídké matice ve formátu CSC.



■ **Obrázek 6.6** Příklad reprezentace řídké matice ve formátu BCSR.

6.1.6 Block Compressed Sparse Row

Formát blokových komprimovaných řádků (z angl. *blocked compressed sparse row*, zkr. BCSR) je blokovou variantou CSR schématu. BCSR rozděluje řídkou matici na husté bloky o rozměrech $r \times c$ a ukládá každý blok, který obsahuje alespoň jeden nenulový prvek, ve formátu CSR. Na každý blok rozdělené matice je tedy nahlíženo jako na jeden ucelený prvek, přičemž tato matice ucelených prvků je pak reprezentována ve formátu CSR. [36, 38]

Blok obsahující pouze nulové prvky není ukládán, a bloky obsahující alespoň jeden nenulový prvek jsou uloženy po dílcích řádcích včetně všech svých nulových prvků. Vstupní matice $\mathbf{A} \in \mathbb{R}^{m,n}$ je kvůli rozdělení na bloky v tomto případě považována za matici o rozměrech $\lceil \frac{m}{r} \rceil \times \lceil \frac{n}{c} \rceil$. Stejně jako CSR schéma i BCSR formát využívá pro reprezentaci matice $\mathbf{A}$ pole pro data, sloupcové indexy a řádkové ukazatele. [36, 38] Příklad reprezentace řídké matice ve formátu BCSR pro $r = c = 2$ je k vidění na obrázku 6.6.

Protože BCSR ukládá bloky ve formátu CSR, odvíjí se od formátu komprimovaných řádků i jeho paměťová náročnost. Pro bloky, které mají rozměr

$r \times c$, a matici $\mathbf{A}$ s celkem nnz_b nenulovými bloky, je paměťová náročnost BCSR následující:

$$S_{BCSR} = \left(\left\lceil \frac{m}{r} \right\rceil + 1 \right) \cdot S_{idx} + nnz_b \cdot (r \cdot c \cdot S_{val} + S_{idx}),$$

kde S_{val} reprezentuje počet bajtů pro uložení nenulového prvku a S_{idx} paměťové nároky na uložení celého čísla.

BCSR sice vyžaduje méně paměti pro pole řádkových ukazatelů a sloupcových indexů ve srovnání s formátem CSR, ale explicitní ukládání nulových prvků v poli dat naopak zvyšuje paměťovou náročnost a navíc se kvůli nulovým prvkům také zvyšuje počet zbytečných přístupů do paměti. Na příkladu BCSR reprezentace na obrázku 6.6 je vidět, že velikost polí řádkových ukazatelů a sloupcových indexů je přibližně o polovinu menší, než kdyby byla matice reprezentována v CSR formátu, pokud budou uvažovány stejné paměťové nároky pro S_{idx} . Nicméně pole dat obsahuje 7 zbytečných nulových prvků, a toto pole je proto o přibližně polovinu větší než při CSR reprezentaci. [36]

Je tedy zřejmé, že volba velikosti bloku je kritická pro výkon algoritmů reprezentujících řídké matice v BCSR formátu. Z toho plynou následující nevýhody:

1. Optimální velikost bloku se liší pro různé řídké matice a BCSR proto na rozdíl od CSR není formátem pro obecné použití. [36]
2. BCSR je užitečný především pro takové řídké matice, které jsou složeny z čtvercových hustých bloků nenulových hodnot, ideálně v nějakém pravidelném vzoru. Nejlepších výsledků dosahuje BCSR na maticích, které jsou rozděleny na velké bloky, čímž se redukuje potřeba ukládat informace o pozici každého nenulového prvku. [36, 38]
3. Velikost hustých bloků musí být často vypočítána dopředu, aby bylo možné zvolit jejich optimální hodnotu, což zvyšuje výpočetní zátěž při reprezentaci matice tímto formátem. [38]
4. Problém s ukládáním nulových prvků v blocích lze vyřešit proměnlivou velikostí jednotlivých bloků. Tento přístup využívá formát VBR (zkr. pro *Variable BCSR*), nicméně nalezení optimálních velikostí nepřekrývajících se bloků je výpočetně velmi náročná úloha. Jedná se dokonce o NP-úplný problém a pro výpočet proměnlivých velikostí bloků jsou proto dostupné pouze heuristické algoritmy. [26, 36, 39]

Z těchto důvodů a pro své omezené použití je BCSR formát jedním z nejméně studovaných formátů pro násobení řídkých matic. [1]

6.1.7 Další úložné formáty

V předchozí části byly analyzovány nejpoužívanější formáty pro reprezentaci řídkých matic, avšak existuje ještě mnoho dalších zajímavých formátů, z nichž

některé stojí alespoň za krátkou zmínku, ačkoliv tyto další formáty nejsou pro násobení řídkých matic hojně využívány.

Všechny formáty, které byly popsány v předchozí kapitole, patří mezi statické reprezentace, tj. umožňují snadné čtení nenulových prvků matice, avšak přidání nového prvku nebo smazání některého existujícího je komplikovanou operací. Tento problém řeší dynamické reprezentace, mezi které patří např. dynamický řídký formát (z angl. *Dynamic Sparse Format*, zkr. DS). [26]

Existuje celá řada tzv. hybridních formátů, které kombinují vlastnosti dvou a více různých formátů za účelem snížení nadměrné paměťové náročnosti a přebytkového ukládání nulových prvků. Např. formát HYB kombinuje formáty COO a ELL tím, že reprezentuje prvních k nenulových prvků každého řádku v ELL formátu a pro zbylé nenulové prvky pak využívá formát COO. [36]

Existuje několik formátů, které mají za cíl efektivně komprimovat velikost indexových polí tradičních formátů. Např. DCSR formát využívá delta kódování pro efektivní reprezentaci rozdílů mezi sloupcovými indexy sousedních nenulových prvků v rámci jednoho řádku a také kombinuje nenulové prvky na po sobě jdoucích pozicích do jedné entity. [36]

Podobně jako BCSR formát rozděluje matici do dvourozměrných bloků, existují formáty, které vstupní matici rozdělují pouze podél jedné její dimenze. Např. formát SELL (zkr. pro *Sliced ELL*) rozděluje matici horizontálně na dvě části, přičemž každou z nich pak reprezentuje ELL formátem. Díky tomuto rozdělení je možné dosáhnout značného snížení ukládání nulových prvků pro účely zarovnání oproti klasickému ELL formátu. [36]

Na závěr stojí za zmínku vysoce specializované formáty, mezi které patří např. CSR5. Tento relativně nový formát byl vyvinut speciálně pro velmi rychlé násobení řídké matice hustým vektorem na vektorových výpočetních architekturách. Tento formát organizuje nenulové prvky matice do tzv. dlaždic fixní velikosti, přičemž nenulové prvky v každé dlaždici jsou uloženy do sloupců, což zajišťuje souvislé vzorce přístupu do paměti na vektorových výpočetních architekturách. Oproti klasickému CSR formátu potřebuje CSR5 schéma kvůli rozdělení nenulových prvků do dlaždic uchovávat pomocné informace v dalších polích, jako jsou ukazatele na dlaždice a deskriptory dlaždic, které obsahují pomocná metadata jako je částečný součet. Při násobení řídké matice hustým vektorem dosahuje tento formát vynikajících výsledků pro takové matice, které mají velmi nepravidelnou řídkou strukturu. [36, 40]

6.2 Souborové formáty řídkých matic

V předchozí kapitole byly popsány formáty pro reprezentaci řídkých matic v paměti počítače. Řídké matice je však také potřeba archivovat a sdílet mezi výzkumnými komunitami. Pro tyto účely vzniklo několik formátů pro ukládání řídkých matic do souboru počítače, které jsou popsány a analyzovány v této kapitole.

6.2.1 Harwell–Boeing

Jedním z prvních souborových formátů pro sdílení řídkých matic je formát Harwell–Boeing. Tento formát vznikl jako součást kolekce řídkých matic z [41] a byl vyvinut speciálně pro snadnou manipulaci v jazyce Fortran, který pro čtení a zápis dat používal způsob založený na sloupcích, kde je pozice dat v každém řádku předem pevně určena. Z toho důvodu jsou data a informace o matici ve formátu Harwell–Boeing organizovány do pevné délky 80 znaků na řádek. Např. prvních 72 znaků prvního řádku hlavičkového bloku je rezervováno pro název matice, zbylých 8 znaků je rezervováno pro identifikátor matice. Dále např. prvních 14 znaků druhé řádky hlavičkového bloku je rezervováno pro reprezentaci celkového počtu řádků v souboru, dalších 14 znaků druhé řádky hlavičkového bloku je rezervováno pro počet řádků, které obsahují sloupcové ukazatele atd. Kromě samotné matice umožňuje formát Harwell–Boeing ukládat i husté vektory pravých stran pro řešení rovnic typu $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. [41]

Každá matice začíná víceřádkovým hlavičkovým blokem, následovaným dvěma až čtyřmi datovými bloky. Hlavičkový blok obsahuje souhrnné informace o reprezentované matici, jako např. název a identifikátor matice, celkový počet řádků v souboru a počet řádků pro jednotlivé datové bloky, typ matice, její dimenze a také formát a velikost její reprezentace, aby bylo možné předem určit, kolik paměti bude reprezentace matice vyžadovat. Harwell–Boeing používá pro reprezentaci matice v paměti schéma CSC nebo formát *Finite Element Matrix Storage*, jenž je určen speciálně pro matice, které jsou výsledkem metody konečných prvků. [41]

Po hlavičkovém bloku následuje datový blok pro sloupcové ukazatele, dále datový blok řádkových indexů a následně datový blok obsahující samotné nenulové prvky matice. Pokud má daný soubor uchovávat i vektory pravých stran, je v souboru ještě čtvrtý datový blok, ve kterém jsou uloženy prvky vektorů pravých stran. [41]

Z popisu je zřejmé, že tento formát je pro člověka velmi špatně čitelný, což v kombinaci s tím, že byl navržen a optimalizován pro použití v jazyce Fortran, má za následek, že se tento formát v dnešní době již téměř nepoužívá. Nicméně vyjma kolekce [41] jsou v tomto formátu k dispozici např. i matice z kolekce [42].

6.2.2 Rutherford–Boeing

Formát Rutherford–Boeing je rozšířenou a vylepšenou verzí formátu Harwell–Boeing a zároveň zachovává zpětnou kompatibilitu. Rutherford–Boeing vznikl jako součást kolekce [43] a přidává podporu pro další metadata, pomocná data a vylepšenou dokumentaci vlastností reprezentované matice. [43]

Nejvýznamnějším vylepšením je způsob zpracování metadat, tj. dodatečných nebo doplňkových informací. Předchozí Harwell–Boeing formát vyžadoval, aby všechny dodatečné informace byly zahrnuty do stejného souboru jako

	1	2	3	4	5
1	1	0	0	6	0
2	0	10,5	0	0	0
3	0	0	0,015	0	0
4	0	250,5	0	-280	33,32
5	0	0	0	0	12

■ **Obrázek 6.7** Příklad obecné čtvercové řídké matice řádu 5 s 8 nenulovými reálnými prvky. [44]

samotná řídká matice, zatímco Rutherford–Boeing umožňuje ukládat dodatečná data do samostatných souborů. Dále tento formát také přináší vylepšenou podporu pro ukládání různých typů matic, jako jsou symetrické nebo Hermitovské matice. Například pro symetrické matice umožňuje formát ukládat pouze nenulové prvky pod resp. nad hlavní diagonálou společně s nenulovými prvky na této hlavní diagonále, neboť zbylé prvky jsou ze symetrie matice jednoznačně určeny. Navíc formát Rutherford–Boeing umožňuje na rozdíl od svého předchůdce ukládat vektory pravých stran nejen v hustém, ale také v řídkém formátu. [43]

Velmi špatná čitelnost pro člověka formátu Rutherford–Boeing v kombinaci se strukturou zaměřenou na jazyk Fortran způsobuje klesající využití ve výzkumu a v dnešní době omezuje i praktickou využitelnost tohoto formátu. Avšak kromě kolekce [43] jsou v tomto formátu dostupné např. i matice z kolekce [24].

6.2.3 Matrix Market

Formát Matrix Market, který byl vyvinut v americkém Národním institutu standardů a technologie jako součást kolekce [42], byl navržen s důrazem na jednoduchost, rozšiřitelnost, snadný parsing a dobrou čitelnost pro člověka.

Soubor ve formátu Matrix Market začíná hlavičkovým řádkem, za kterým následuje libovolný počet řádků s komentáři, přičemž každý řádek s komentářem musí začínat symbolem `%`. Po komentářích následuje řádek obsahující rozměry matice a počet jejích nenulových prvků. Za tímto řádkem pak následují samotná data o nenulových prvcích matice. [44]

Hlavičkový řádek začíná klíčovým identifikátorem `%MatrixMarket`, za kterým následuje definice objektu, který je v souboru reprezentován, konkrétně

```
%%MatrixMarket matrix coordinate real general
%=====
%
% This ASCII file represents a sparse 5x5
% matrix with 8 nonzeros.
%
%=====
5 5 8
1 1 1.000e+00
2 2 1.050e+01
3 3 1.500e-02
1 4 6.000e+00
4 2 2.505e+02
4 4 -2.800e+02
4 5 3.332e+01
5 5 1.200e+01
```

■ **Výpis kódu 6.1** Reprezentace matice z obrázku 6.7 ve formátu Matrix Market. [44]

matice nebo vektoru. Následuje specifikace reprezentace daného objektu, přičemž je podporován souřadnicový formát COO nebo hustý formát. Poté následuje určení datového typu nenulových prvků, přičemž možné typy jsou reálný, komplexní, celočíselný nebo binární. Na závěr je specifikována symetrie matice, která může být obecná, symetrická, antisymetrická nebo Hermitovská. V případě symetrických matic je možné stejně jako u formátu Rutherford–Boeing ukládat pouze nenulové prvky z dolní, resp. horní trojúhelníkové části této řídké matice, neboť ostatní prvky jsou ze symetrie matice určeny jednoznačně. [44]

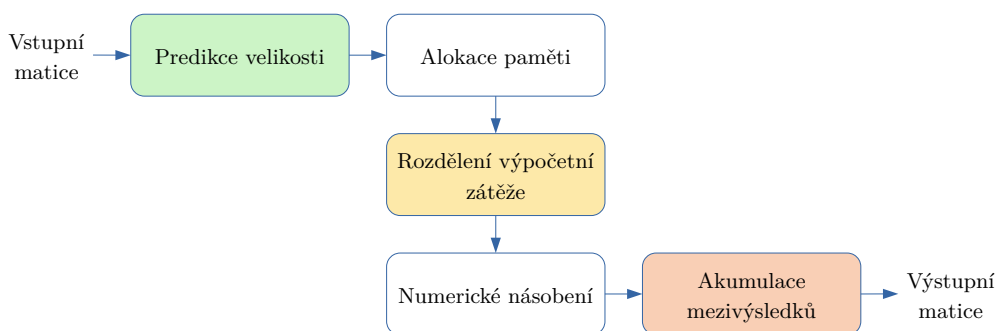
Obrázek 6.7 ukazuje obecnou čtvercovou matici řádu 5 s 8 nenulovými reálnými prvky. Reprezentaci této řídké matice v souborovém formátu Matrix Market pomocí koordinačního schématu pak ukazuje výpis 6.1.

Díky své jednoduchosti a srozumitelnosti se Matrix Market stal ve vědecké komunitě rozšířeným a preferovaným formátem pro sdílení datových sad řídkých matic. Například veškeré matice z kolekcí [24] a [42] jsou dostupné právě ve formátu Matrix Market.

Algoritmy násobení řídkých matic

Tato kapitola rozebírá obecný postup výpočtu SpGEMM algoritmů, detailně pak analyzuje, jak dílčí části tohoto postupu řeší existující algoritmy pro násobení řídkých matic.

Algoritmy pro násobení řídkých matic jsou nesmírně komplexní a skládají se z několika dílčích algoritmů. Typický postup výpočtu SpGEMM algoritmů, který znázorňuje obrázek 7.1, má celkem pět dílčích fází. Fáze predikce velikosti se zabývá předpovědí paměťové náročnosti výstupní matice ještě před spuštěním samotného násobení vstupních matic. Následuje fáze alokace paměti, která využívá výsledek predikce velikosti výstupní matice z předchozí fáze k vyhrazení dostatečného paměťového prostoru, který bude potřebný pro numerické násobení vstupních matic. Cílem třetí fáze je efektivní rozdělení výpočetní zátěže. Samotné násobení vstupních matic se provádí ve čtvrté fázi, ve které může vznikat větší množství částečných výsledků. Akumulace těchto mezivýsledků a vypočítání konečného výsledku je pak úkolem poslední páté fáze. [1]



■ **Obrázek 7.1** Obecný postup výpočtu SpGEMM algoritmů. Překresleno z [1].

Vzhledem k náročnosti a komplexnosti SpGEMM algoritmů není výjimkou, že se určité vědecké články věnují pouze některým z dílčích fází SpGEMM algoritmu. Například algoritmy SaSpGEMM a MAGNUS publikované ve článcích [45] a [5] se soustředí na návrh efektivního akumulátoru mezivýsledků, přičemž další fáze nejsou téměř vůbec probírány. Z těchto důvodů bude tato kapitola strukturována podle dílčích fází SpGEMM algoritmu.

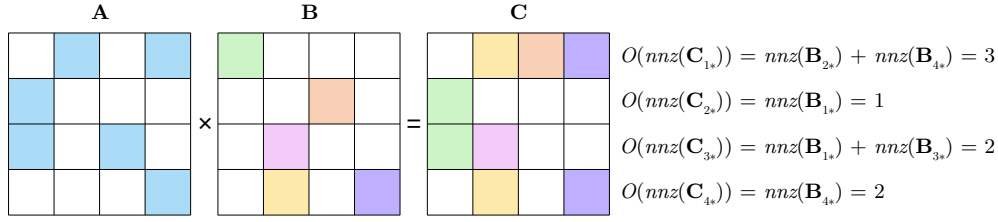
Fáze alokace paměti se řeší pomocí konstruktů pro alokaci paměti v programovacím jazyce, který je použit pro implementaci SpGEMM algoritmu. Fáze numerického násobení pak využívá jednu ze čtyř formulací pro násobení obecných matic, které byly popsány a analyzovány v kapitole 5, přičemž většina algoritmů využívá RbR formulaci. Tato kapitola se proto bude věnovat zbývajícím třem fázím, přičemž pro každou z nich budou popsány a analyzovány existující metody, které je řeší.

7.1 Predikce velikosti výstupní řídké matice

Jak plyne z definice 5.8, výstupem násobení řídkých matic $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, kde $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,p}$, $\mathbf{C} \in \mathbb{R}^{m,p}$, je také řídká matice, která musí být v paměti reprezentována některým z formátů, které byly popsány v kapitole 6. Jak bylo v této kapitole zmíněno, nejčastější volbou pro reprezentaci matice je CSR formát, avšak bez ohledu na zvolený formát bude významnou složkou paměťové náročnosti uložení samotných nenulových prvků výstupní matice. Nicméně tento počet nenulových prvků včetně jejich rozmístění ve výstupní matici nelze určit předem bez provedení samotného násobení. Proto způsob predikce počtu nenulových prvků výstupní matice významně ovlivňuje efektivní správu paměti SpGEMM algoritmu, neboť příliš benevolentní odhad může vést ke zbytečnému plýtvání pamětí nebo dokonce ke zhoršení principu prostorové lokality cache paměti (viz. kapitola 2.2), a nedostatečný odhad může naopak vyžadovat časté realokace paměti a kopírování či přesouvání již vypočtených dat. [1]

7.1.1 Metoda horního odhadu

Jedním ze způsobů predikce počtu nenulových prvků výstupní matice je stanovení hodnoty, pro kterou platí, že počet nenulových prvků nemůže být vyšší než tato hodnota. Tento vypočtený údaj se nazývá horní odhad. Cílem predikce pomocí metody horního odhadu (v angl. *upper-bound prediction*) je nejen stanovení bezpečného odhadu počtu nenulových prvků, který zajistí, že během numerického násobení nebude potřeba provádět žádné realokace paměti, ale také to, aby stanovený odhad nadhodnocoval počet nenulových prvků co nejméně a tím se zabránilo nadměrnému plýtvání pamětí. Existuje proto několik způsobů, jak horní odhad stanovit. Horní odhad počtu nenulových prvků výstupní matice bude dále označován také jako $O(nnz(\mathbf{C}))$.



■ **Obrázek 7.2** Ukázka použití metody horního a přesného odhadu počtu nenulových prvků matice **C**. Překresleno z [1].

Očividný způsob stanovení horního odhadu je pomocí rozměrů výstupní matice **C**, tj. $O(nnz(\mathbf{C})) = m \cdot p$. Je zřejmé, že tento odhad je velmi hrubý, protože fakticky ignoruje řídkost matice a na výstupu předpokládá hustou matici, která nemá žádné nulové prvky. Tento odhad je proto nevhodný a v praxi navíc špatně využitelný, protože pro matici velkých rozměrů může být nereálné alokovat paměť pro všechny její pozice.

Jako další odhad se nabízí součin počtu nenulových prvků vstupních matic, tj. $O(nnz(\mathbf{C})) = \min(nnz(\mathbf{A}) \cdot nnz(\mathbf{B}), m \cdot p)$. Tento odhad sice může být v některých případech jemnější než předchozí, stále je však nevhodný kvůli významnému plýtvání pamětí, které je obzvláště problematické pro matice velkých rozměrů.

Významně přesnější odhad je možné získat vhodnou analýzou struktury vstupních matic, která využívá principu RbR násobení matic. Pro každý i -tý řádek matice **C** totiž platí, že počet nenulových prvků v tomto řádku nemůže být vyšší, než je součet nenulových prvků ve všech řádcích matice **B**, které odpovídají každému nenulovému prvku v i -tém řádku matice **A**. [1, 46]

Pro i -tý řádek matice **C** tedy platí následující odhad:

$$O(nnz(\mathbf{C}_{i*})) = \min(p, \sum_{k \in \mathcal{J}(\mathbf{A}_{i*})} nnz(\mathbf{B}_{k*})),$$

kde $\mathcal{J}$ značí množinu sloupcových indexů nenulových prvků dané matice, tj. $\mathcal{J}(\mathbf{A}) = \{k \in \{1, \dots, n\} \mid A_{ik} \neq 0 \wedge i \in \{1, \dots, m\}\}$ pro $\mathbf{A} \in \mathbb{R}^{m,n}$. [46]

Princip tohoto horního odhadu ukazuje obrázek 7.2. První řádek matice **A** má dva nenulové prvky, jejichž sloupcové indexy jsou 2 a 4. Horní odhad počtu nenulových prvků v prvním řádku výstupní matice **C** se proto rovná součtu nenulových prvků ve druhém a čtvrtém řádku matice **B**. [1]

Metoda horního odhadu zaručuje, že přidělená paměť bude vždy dostatečná, čímž je eliminováno riziko realokace během numerického násobení. Další výhodou této metody je poměrně nízká časová složitost $O(nnz(\mathbf{A}))$, neboť vyžaduje pouze iteraci přes nenulové prvky matice **A** a součet počtu nenulových prvků v odpovídajících řádcích matice **B**, což při reprezentaci ve formátu CSR lze vypočítat v čase $O(1)$. Nevýhodou této metody je, že může vést k nadbytečnému alokování paměti, která ze značné části nebude při výpočtu vůbec využita. Další nevýhodou je, že při použití horního odhadu je nutné při nume-

rickém násobení pracovat s dočasnou maticí, pro kterou je paměť alokována pomocí horního odhadu, a teprve po skončení numerického násobení je možné alokovat paměť pro výstupní matici na základě skutečného počtu nenulových prvků, přičemž vypočtená data pak musí být z dočasné matice přesunuta do skutečné výstupní matice v požadovaném formátu. [46, 47, 48]

Metodu horního odhadu využívá například algoritmus ESC [46] nebo algoritmus BRMerge [48], který byl implementován ve dvou variantách, z nichž jedna používá horní odhad a druhá přesnou alokaci (viz. 7.1.2). Dalším reprezentantem této metody je algoritmus bhSPARSE [47], který však po výpočtu horního odhadu používá pro alokaci paměti hybridní způsob, jenž přiděluje paměť pro kratší řádky matice $\mathbf{C}$ na základě výsledku horního odhadu a pro delší řádky alokuje paměť pomocí progresivní metody (viz. 7.1.4).

7.1.2 Metoda přesné alokace

Cílem této metody je určit přesný počet nenulových prvků včetně jejich konkrétního rozložení ve výstupní matici $\mathbf{C}$. Tato metoda provádí logiku násobení matic, avšak bez provedení skutečných aritmetických operací nad nenulovými prvky vstupních matic. Proto se tato metoda často označuje také jako symbolická fáze. V symbolické fázi je tedy vypočítán přesný počet nenulových prvků v každém řádku výstupní matice na základě indexů řádků a sloupců vstupních matic. Skutečné hodnoty nenulových prvků jsou vypočítány až v numerické fázi. Po symbolické fázi lze přesně alokovat paměť prostřednictvím prefixového součtu na základě vypočítaných informací o počtu a rozmístění nenulových prvků. [1, 47, 48]

Na rozdíl od metody horního odhadu musí tento způsob zohlednit překrývání sloupcových indexů ve vstupní matici $\mathbf{B}$, a tím zajistit, že pokud více součinů nenulových prvků spadá pod stejný sloupcový index v daném řádku výstupní matice $\mathbf{C}$, bude alokována paměť pouze pro jeden takový prvek. Pro i -tý řádek matice $\mathbf{C}$ je tedy počet nenulových prvků vypočítán následovně:

$$nnz(\mathbf{C}_{i*}) = \left| \bigcup_{k \in \mathcal{J}(\mathbf{A}_{i*})} \mathcal{J}(\mathbf{B}_{k*}) \right|,$$

kde $\mathcal{J}$ opět značí množinu sloupcových indexů nenulových prvků dané matice. Příklad přesné alokace ukazuje obrázek 7.2, kde jsou nenulové prvky matic vyobrazeny jako barevná pole jednotlivých matic.

Je zřejmé, že vzhledem k tomu, že symbolická fáze neprovádí aritmetické operace se skutečnými hodnotami nenulových prvků, stále může pro některou pozici výstupní matice dojít k algebraickému vynulování, tj. situaci, kdy součtem dílčích mezivýsledků bude nula. Tuto situaci je však nemožné předpovědět dopředu bez provedení samotného numerického násobení. Nevýhodou tedy je, že i metoda přesné alokace může někdy paměťové nároky mírně nadhodnocovat. Jak se píše v knize [23], je v takových situacích přijatelné nechat

tyto nulové prvky uložené ve výsledné matici. Alternativou je implementace operace prořezávání, která po numerické fázi dodatečně zkontroluje, zda při násobení nebyly do výsledné matice uloženy nulové prvky, a tyto nulové prvky odstraní.

Další nevýhodou této metody je její velmi vysoká časová složitost, která je srovnatelná s numerickým násobením vstupních řídkých matic. Doba běhu algoritmu, který tuto metodu využívá, se proto použitím této metody může zdvojnásobit, neboť operaci násobení řídkých matic v takovém případě algoritmus fakticky provádí dvakrát. Z tohoto důvodu je tato metoda výhodná tam, kde se opakuje stejná struktura rozložení nenulových prvků, akorát s jinými číselnými hodnotami, jako např. v iterativních metodách. [47, 48]

Významnou výhodou této metody však je, že poskytuje přesný počet nenulových prvků ve výstupní matici $\mathbf{C}$, vyjma výjimečných případů algebraického vynulování. Díky tomu je možné metodu přesné alokace využít k optimální alokaci paměťového prostoru pro výstupní matici bez nadměrného plýtvání pamětí nebo četných realokací, které mohou nastat v případě podhodnocení počtu nenulových prvků, jako je tomu u pravděpodobnostní metody a progresivní metody (viz. dále). [48]

Metodu přesné alokace používá například algoritmus RMerge [49], a dále ji používají také SpGEMM algoritmy, které jsou dostupné v knihovnách Intel MKL a Nvidia cuSPARSE. [1, 47]

7.1.3 Pravděpodobnostní metoda

Pravděpodobnostní metody odhadují počet nenulových prvků ve výstupní matici pomocí statistického modelování, které je založeno na vlastnostech rozložení nenulových prvků v obou vstupních maticích. Tyto metody tedy místo explicitního výpočtu počtu nenulových prvků nahlíží na přítomnost nenulového prvku ve výstupní matici jako na náhodný jev. Tyto metody většinou na základě podmnožiny řádků a sloupců vstupních matic analyzují určitou část potenciální výstupní matice a následně na základě této analýzy odhadnou celkový počet nenulových prvků ve výstupní matici. [1, 50, 51]

Například metoda publikovaná ve článku [51] nejprve vytvoří matici $\mathbf{A}'$ tak, že náhodně vybere podíl p řádků vstupní matice $\mathbf{A}$, kde $p \in (0, 1)$. Stejným způsobem je zkonstruována matice $\mathbf{B}'$ výběrem náhodného podílu p sloupců matice $\mathbf{B}$. Následně metoda odhadne $nnz(\mathbf{A}' \cdot \mathbf{B}')$ pomocí všech dílčích součinů nenulových prvků vstupních matic. Pro každý dílčí součin $\mathbf{A}'_{ik} \cdot \mathbf{B}'_{kj}$ je prostřednictvím hashovací funkce převedena dvojice indexů (i, k) na hodnotu z intervalu $[0, 1]$. Počet nenulových prvků maticového součinu $nnz(\mathbf{A}' \cdot \mathbf{B}')$ je pak následně odhadnut jako $\frac{l}{v}$, kde v je l -tá nejmenší zahashovaná hodnota, pro určité pevně zvolené l . Odhad celkového počtu nenulových prvků ve výstupní matici $\mathbf{C}$ je pak vypočítán jako $\frac{l}{vp^2}$. Článek [52] experimentálně vyhodnotil, že nejlepší výsledky je dosaženo při volbě $l = 100$ a $p = \frac{1}{10}$. Dále autoři tohoto článku pro odstranění nedeterminismu a zajištění reprodukovatelnosti nevolí

řádky matice $\mathbf{A}$, resp. sloupce matice $\mathbf{B}$ náhodně, ale systematicky vybírají každý desátý řádek matice $\mathbf{A}$, resp. sloupec matice $\mathbf{B}$. [51, 52]

Výhodou pravděpodobnostních metod je jejich nízká časová složitost, avšak tyto metody poskytují dobré výsledky především pro matice, které mají poměrně rovnoměrné rozložení nenulových prvků, a mohou poskytovat špatné výsledky pro matice, které mají nerovnoměrnou strukturu nenulových prvků. Další nevýhodou je, že pokud je získaný odhad příliš nízký, bude nutné během výpočtu alokovat nový paměťový blok a zkopírovat či přesunout již vypočtená data. V případě velmi podhodnoceného odhadu může být během numerické fáze potřeba provést realokaci paměti dokonce několikrát. Navíc, stejně jako u metody horního odhadu, je v případě použití pravděpodobnostních metod nutné při numerické fázi pracovat v pomocném paměťovém prostoru, protože alokovat paměť pro výstupní matici v požadovaném formátu je možné až poté, co je přesně znám počet nenulových prvků matice $\mathbf{C}$.

Kvůli uvedeným nevýhodám není tato metoda často používána, nicméně byla využita např. při implementaci SpGEMM algoritmu ve článku [52].

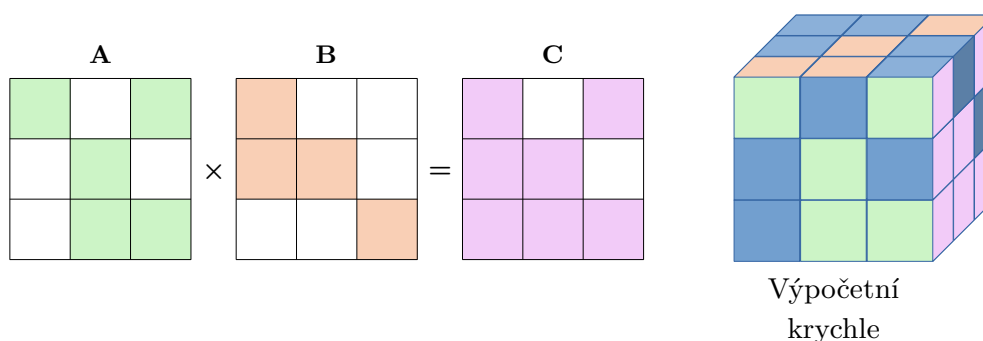
7.1.4 Progresivní metoda

Progresivní metoda se zásadně liší od předchozích tří metod. Namísto výpočtu velikosti výstupní matice ještě před numerickou fází zpracovává tato metoda odhad velikosti dynamicky během samotného násobení vstupních matic. Z tohoto důvodu se jedná spíše o algoritmickou strategii než o predikční výpočet. [1]

Algoritmy používající tento způsob nejprve před numerickou fází odhadnou počáteční velikost a poté spustí násobení vstupních matic. Pokud nastane situace, že je aktuální blok paměti nedostatečný, je alokovan nový paměťový blok, který je větší než aktuální o určitý konstantní faktor, jako např. o 1,5, a již vypočtená data se zkopírují nebo přesunou do tohoto nově alokovaného bloku. [1, 47, 53]

Výhodou této metody je, že se dokáže velmi dobře adaptovat na rozličné struktury rozmístění nenulových prvků výstupní matice. Progresivní metoda také nepředstavuje žádnou výpočetní zátěž před spuštěním numerické fáze algoritmu, nicméně tuto zátěž nahrazuje režii dynamické správy paměti. Významnou nevýhodou progresivní metody je, že každá realokace vyžaduje zkopírování či přesunutí všech již vypočtených dat, což může daný SpGEMM algoritmus výrazně zpomalovat, zejména pokud výstupní matice dosahuje vysokých rozměrů. Další nevýhodou je, že stejně jako u horního odhadu nebo pravděpodobnostní metody je při použití progresivní metody nutné v numerické fázi pracovat s pomocnou maticí, resp. pomocným paměťovým prostorem, ze kterého jsou vypočtená data zkopírována do výsledné matice v požadovaném formátu až po skončení numerické fáze na základě přesného počtu vypočtených nenulových prvků. [1, 47]

Tuto metodu používá např. implementace SpGEMM algoritmu v prostředí MATLAB [53]. Dalším reprezentantem je i algoritmus bhSPARSE [47], který



■ **Obrázek 7.3** Ukázka reprezentace násobení matic prostřednictvím výpočetní krychle. Inspirováno ilustrací ze článku [1].

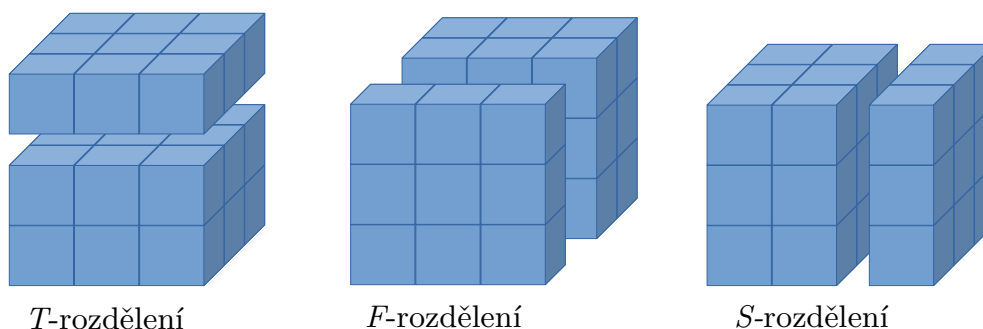
na základě horního odhadu identifikuje kratší a delší řádky výstupní matice. Pro kratší řádky alokuje bhSPARSE paměť ještě před násobením matic na základě výsledku horního odhadu, pro delší řádky tento algoritmus alokuje paměť dynamicky při numerické fázi pomocí progresivní metody. [1, 47]

7.2 Rozdělení výpočetní zátěže

Na rozdíl od násobení hustých matic nelze výpočetní zátěž SpGEMM násobení předpovědět předem, neboť počet nenulových prvků včetně jejich rozložení ve výstupní matici je neznámý bez provedení numerického násobení. Pro efektivní paralelizaci SpGEMM násobení je proto nezbytné sofistikované rozdělení výpočtu mezi procesorové jednotky, aniž by došlo k výrazně nerovnoměrnému zatížení či vysoké synchronizační režii. Metody pro rozdělení výpočtu se řadí do dvou kategorií, které jsou blokové a grafové rozdělení výpočtu. Volba metody rozdělení výpočtu určuje granularitu, s jakou je výpočet rozdělen mezi jednotlivá vlákna, přičemž výpočet může být rozdělen na úrovni řádků, sloupců či jednotlivých prvků výstupní matice, nebo na úrovni výpočtu mezivýsledků, které je potřeba následně sloučit do finálního výsledku.

7.2.1 Blokové rozdělení výpočetní zátěže

Násobení řídkých matic lze geometricky interpretovat jako krychli, která se skládá z jednotlivých podkrychlí (případně také voxelů), které odpovídají vynásobení dvou nenulových prvků vstupních matic **A** a **B**. Tato geometrická reprezentace se nazývá výpočetní krychle (v angl. *work cube* nebo také *computation cube*), jejíž ukázka pro vstupní matice řádu 3 je k vidění na obrázku 7.3. Přední a horní pohled na výpočetní krychli představují vstupní matice SpGEMM násobení, boční pohled pak znázorňuje výstupní matici. Na základě této reprezentace lze na rozdělení výpočtu násobení řídkých matic nahlížet jako na rozdělení výpočetní krychle v různých dimenzích. Blokové metody pro



■ **Obrázek 7.4** Tři varianty jednorozměrného rozdělení výpočetní krychle.
Překresleno z [1].

rozdělení výpočetní zátěže lze proto klasifikovat na jednorozměrné, dvourozměrné a trojrozměrné na základě toho, jak jsou jednotlivé voxely výpočetní krychle přiřazovány výpočetním vláknům. [1, 54, 55]

Jednorozměrné (nebo také 1D) metody rozdělují výpočetní krychli pouze v jedné ze tří dimenzí. Všechny tři varianty jednorozměrného rozdělení ukazuje obrázek 7.4. Pro každou variantu platí, že daná vrstva výpočetní krychle je přiřazena jedné výpočetní jednotce. [1, 54]

T-rozdělení (v angl. *T-partition*) dělí výpočetní krychli rovinami rovnoběžnými s pohledem shora. Jedna jednotka SpGEMM výpočtu proto představuje vynásobení řádku matice **A** a celé matice **B**, přičemž výsledkem je celý řádek výstupní matice **C**. [1, 54]

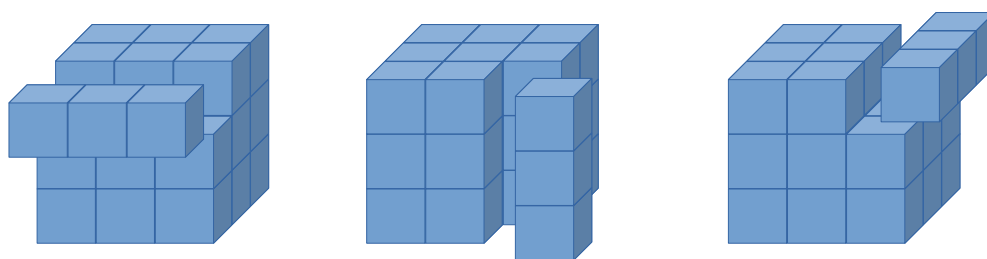
Rozdělení výpočtu dělicí výpočetní krychli rovinami, které jsou paralelní s pohledem zepředu, se nazývá *F*-rozdělení (v angl. *F-partition*). Jedna jednotka SpGEMM výpočtu představuje vynásobení sloupce matice **B** a celé matice **A**. Výsledkem této jednotky SpGEMM výpočtu pak je celý sloupec výstupní matice. [1, 54]

S-rozdělení (v angl. *S-partition*) rozděluje výpočetní krychli rovinami rovnoběžnými z boku. Jedna jednotka SpGEMM výpočtu tak představuje vynásobení sloupce matice **A** a řádku matice **B**. V tomto případě je výstupem pouze částečný výsledek matice **C**. [1, 54]

Dvourozměrné (nebo také 2D) metody rozdělují výpočetní krychli ve dvou ze tří dimenzí, proto i v tomto případě existují celkem tři varianty, které jsou k vidění na obrázku 7.5.

TF-rozdělení dělí výpočetní krychli podle rovin, které jsou rovnoběžné s horním a předním pohledem na tuto krychli. Jeden díl SpGEMM výpočtu tak představuje vynásobení řádku matice **A** a sloupce matice **B**, přičemž výsledkem je právě jeden prvek výstupní matice **C**. Zároveň se jedná o jedinou 2D metodu, jejíž jedna jednotka SpGEMM výpočtu přispívá do výstupní matice celým výsledkem a ne pouze částečným výsledkem. [1, 54]

Rozdělení, které dělí výpočetní krychli v rovinách paralelních s předním a bočním pohledem, se nazývá *FS*-rozdělení. Jedna část SpGEMM výpočtu

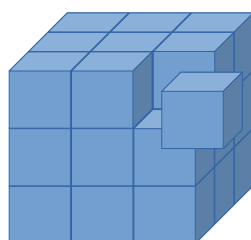


TF-rozdělení

FS-rozdělení

TS-rozdělení

■ **Obrázek 7.5** Tři varianty dvourozměrného rozdělení výpočetní krychle. Překresleno z [1].



3D rozdělení

■ **Obrázek 7.6** Ukázka trojrozměrného rozdělení výpočetní krychle. Překresleno ze článku [1].

představuje vynásobení sloupce matice **A** jedním prvkem matice **B**. Výstupem jedné části výpočtu podle *FS*-rozdělení pak je částečný výsledek jednoho sloupce matice **C**. [1, 54]

Posledním dvourozměrným rozdělením je *TS*-rozdělení, které dělí výpočetní krychli v rovinách, jež jsou rovnoběžné s horním a bočním pohledem na tuto krychli. Jeden díl SpGEMM výpočtu pak představuje vynásobení prvku matice **A** s řádkem matice **B**, přičemž výstupem je částečný výsledek řádku matice **C**. [1, 54]

Trojrozměrné (nebo také 3D) rozdělení dělí výpočetní krychli podle všech tří dimenzí. Ukázka trojrozměrného rozdělení výpočetní krychle je k vidění na obrázku 7.6. Jedna část SpGEMM výpočtu tak představuje vynásobení prvku matice **A** s prvkem matice **B**. [1, 54]

Pro každou blokovou metodu platí, že daná výpočetní jednotka nemusí v jednom kroku obdržet pouze jeden díl k výpočtu podle daného rozdělení, ale může obdržet více těchto částí, tj. může dostat k výpočtu celý blok jednotlivých dílů. [1, 54]

Je zřejmé, že za účelem zmenšení synchronizační režie a komunikace mezi vlákny budou favorizovány metody, jejichž jeden díl výpočtu přispívá do výsledné matice **C** finálním výsledkem a nikoliv pouze mezivýsledkem. Mezi tato rozdělení patří i jednorozměrné *T*-rozdělení, které je zároveň nejpoužívanější blokovou metodou. Mezi další výhody *T*-rozdělení patří, že jeden díl výpo-

čtu podle tohoto rozdělení umožňuje provést jeden krok RbR násobení, které je zároveň nejpoužívanější formulací pro násobení matic (viz. 5). Tím, že jeden díl výpočtu podle T -rozdělení odpovídá řádku matice $\mathbf{A}$ a všem řádkům matice $\mathbf{B}$, je toto rozdělení výhodné pro práci s formátem komprimovaných řádků, který je nejpoužívanějším formátem pro reprezentaci matic v paměti (viz. 6). [1]

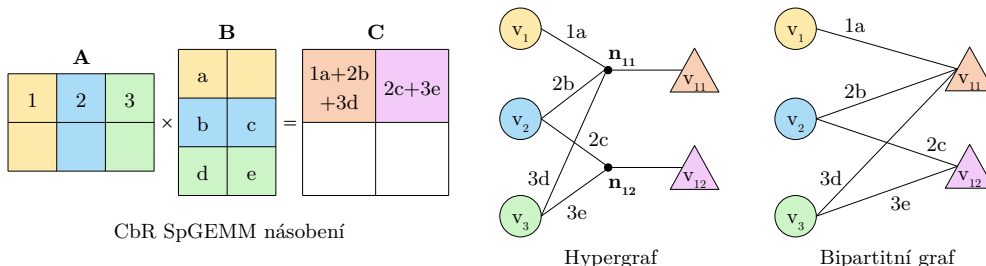
Algoritmy, které SpGEMM násobení paralelizují na grafických kartách (v angl. *graphics processing unit*, zkr. GPU), navíc za účelem efektivnějšího rozdělení výpočetních GPU prostředků kategorizují řádky matice $\mathbf{A}$ do příhrádek na základě očekávané výpočetní zátěže. Například algoritmus ESC [46] nebo algoritmus BRMerge [48] odhadují tuto výpočetní zátěž prostřednictvím celkového počtu násobení nenulových prvků v jednom dílu T -rozdělení. Algoritmus bhSPARSE [47] očekávanou výpočetní zátěž jednoho dílu T -rozdělení určuje pomocí horního odhadu počtu nenulových prvků v řádku výstupní matice $\mathbf{C}$. Další možností, kterou využívá algoritmus RMerge [49], je iterativní rozdělení vstupní matice $\mathbf{A}$ na součin takových matic, aby pro každou matici v tomto součinu platilo, že počet nenulových prvků v každém jejím řádku nepřesáhne hodnotu W , která odpovídá počtu vláken v jedné podskupině GPU vláken (v angl. *sub-warp*).

7.2.2 Grafové rozdělení výpočetní zátěže

Druhý způsob rozdělení SpGEMM výpočtu spočívá v rozkladu grafové reprezentace násobení matic, které je možné reprezentovat prostřednictvím hypergrafu nebo bipartitního grafu. Obě tyto reprezentace budou demonstrovány pro CbR násobení matic. [1]

Hypergraf pro násobení matic $\mathcal{H} = (V, N) = (V^{\mathbf{AB}} \cup V^{\mathbf{C}}, N)$ je definován jako množina vrcholů V a množina hyperhran N , kde každá hyperhrana $n_{ij} \in N$ je podmnožinou množiny vrcholů, tj. $n_{ij} \subseteq V$. Každý vrchol $v_i \in V^{\mathbf{AB}}$ označuje vnější součin sloupce $\mathbf{A}_{*i}$ s řádkem $\mathbf{B}_{i*}$. Množina $V^{\mathbf{C}}$ obsahuje vrchol v_{ij} pro každý nenulový prvek $\mathbf{C}_{ij}$ výstupní matice. Množina hyperhran N pak obsahuje hyperhranu n_{ij} pro každý nenulový prvek $\mathbf{C}_{ij}$. Hyperhrana n_{ij} obsahuje vrchol v_{ij} společně s každým vrcholem v_i , který přispívá některým svým částečným výsledkem do nenulového prvku $\mathbf{C}_{ij}$. Ukázka reprezentace SpGEMM násobení pomocí hypergrafu je k vidění na obrázku 7.7. Cílem rozdělení výpočtu pomocí této reprezentace je rozklad hypergrafu na několik přibližně stejně velkých částí tak, aby byla minimalizována účelová funkce hyperhran, které spojují jednotlivé části hypergrafu po jeho rozkladu. [1, 56, 57]

Bipartitní graf $\mathcal{G} = (V, E) = (V^{\mathbf{AB}} \cup V^{\mathbf{C}}, E)$ pro násobení matic je definován jako dvě disjunktní množiny vrcholů $V^{\mathbf{AB}}$ a $V^{\mathbf{C}}$, a množina hran E , přičemž každá hrana spojuje vrchol z množiny $V^{\mathbf{AB}}$ s vrcholem množiny $V^{\mathbf{C}}$. Význam vrcholů grafu $\mathcal{G}$ je stejný jako v případě hypergrafu. Hrana mezi vrcholem $v_k \in V^{\mathbf{AB}}$ a vrcholem $v_{ij} \in V^{\mathbf{C}}$ existuje právě tehdy, když vnější součin reprezentovaný vrcholem $v_k \in V^{\mathbf{AB}}$ obsahuje částečný výsledek pro ne-



■ **Obrázek 7.7** Ukázka reprezentace SpGEMM násobení matic pomocí hypergrafu a bipartitního grafu. Překresleno z [1].

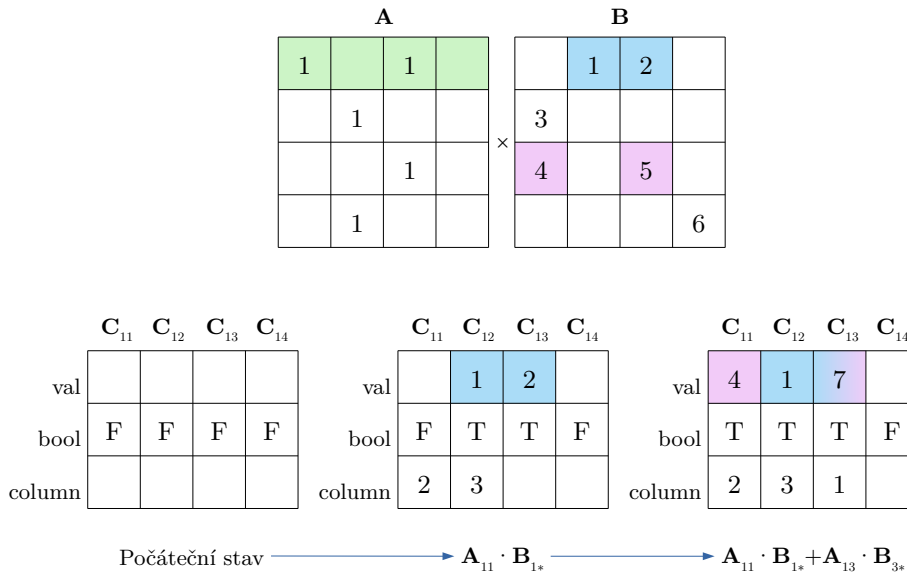
nulový prvek C_{ij} . Příklad reprezentace SpGEMM násobení prostřednictvím bipartitního grafu ukazuje obrázek 7.7. Stejně jako v případě hypergrafu je i v tomto případě cílem rozklad bipartitního grafu na několik podobně velkých částí s minimalizací účelové funkce hyperhran. [1, 56, 57]

Ačkoliv tyto metody umožňují jemnější rozložení výpočetní zátěže, problém rozkladu grafu je výpočetně velmi náročná úloha. Jedná se dokonce o NP-těžký problém, ačkoliv zůstává otevřenou otázkou, zda je rozklad SpGEMM instancí hypergrafu také NP-těžký problém. Pro grafové rozdělení výpočetní zátěže jsou proto dostupné pouze heuristické algoritmy k aproximaci optimálního rozdělení. Z těchto důvodů se grafové rozdělení používá především v distribuovaných systémech, ve kterých je kritické udržet komunikaci mezi jednotlivými uzly výpočetního klastru na co nejnížší úrovni. [56, 57, 58]

7.3 Akumulace mezivýsledků

Zatímco pro násobení hustých matic je pozice pro každý prvek ve výstupní matici předem pevně určená, u násobení řídkých matic není struktura výstupní matice známa předem, neboť výstupní matice je také řídká. Zásadním problémem SpGEMM násobení tedy je, že pozice nenulových prvků ve výstupní matici nejsou známy, dokud není provedeno samotné násobení vstupních matic, případně alespoň jeho symbolická fáze (viz. 7.1.2). Při iterování přes vstupní matice $A \in \mathbb{R}^{m,n}$ a $B \in \mathbb{R}^{n,p}$ jsou násobením jejich nenulových prvků generovány mezivýsledky, které následně musí být agregovány, aby vytvořily konečné výsledky v podobě nenulových prvků výstupní matice $C \in \mathbb{R}^{m,p}$. Výkon SpGEMM algoritmu proto do značné míry závisí na tom, jakým způsobem algoritmus mezivýsledky spravuje a zpracovává. Mechanismus, který je odpovědný za ukládání, správu a sčítání těchto dílčích výsledků, se nazývá akumulátor (z angl. *accumulator*).

V kontextu RbR násobení, které je pro SpGEMM nejpoužívanější formulací násobení matic (viz. kapitola 5), představuje výpočet jednoho řádku výstupní matice lineární kombinaci řádků matice B , přičemž koeficienty této lineární kombinace jsou určeny nenulovými prvky z daného řádku matice A . Vzhledem



■ **Obrázek 7.8** Ukázka použití SPA akumulátoru. Překresleno z [1].

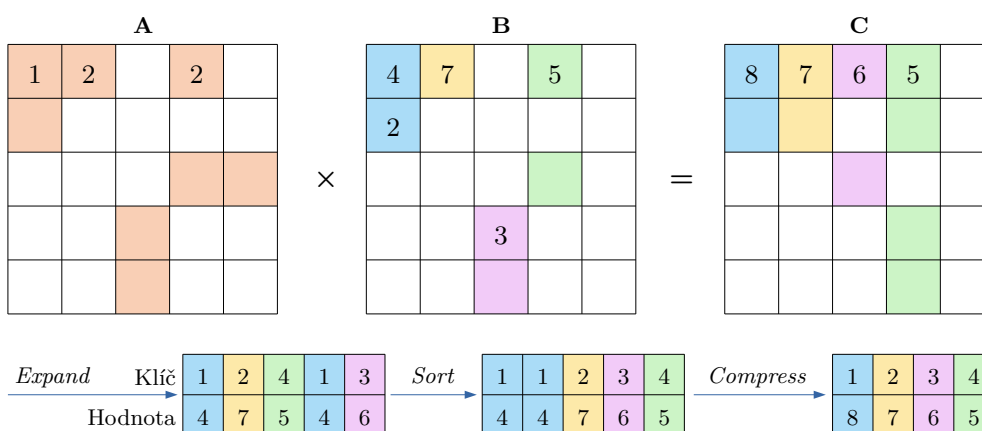
k tomu, že rozložení nenulových prvků v řádku C_{i*} odpovídá sjednocení pozic nenulových prvků z přispívajících řádků matice **B**, musí akumulátor určit, které indexy v řádku C_{i*} budou obsahovat nenulové prvky a sečíst hodnoty, které jsou spojeny s těmito indexy. Akumulátor také poskytuje dočasný paměťový prostor, ve kterém jsou tyto operace prováděny, než je výstupní řádek zapsán do paměťového prostoru určeného pro požadovaný formát matice **C**, kterým je nejčastěji formát komprimovaných řádků CSR (viz. 6). [1]

Pro účely analýzy existujících metod pro akumulaci mezivýsledků bude v této části předpokládáno, že je použit SpGEMM algoritmus, který využívá RbR formulaci násobení matic a pro reprezentaci matic používá formát komprimovaných řádků CSR.

7.3.1 SPA

Nejjednodušší způsob akumulace mezivýsledků je hustý akumulátor, který používá husté pole o velikosti počtu sloupců matice **C** k ukládání dílčích výsledků, a pole příznaků k ověření, zda byl na daný sloupcový index již uložen dílčí výsledek. Některé implementace, jako je např. SPA v prostředí MATLAB, navíc používají neuspořádaný seznam sloupcových indexů, jejichž příznak je nastaven na hodnotu `true`. Ukázka použití tohoto akumulátoru je k vidění na obrázku 7.8. [1, 53, 59]

Tento akumulátor poprvé představil Gustavson ve svém článku [59], později byl tento akumulátor formalizován ve článku [53] pod názvem řídký akumulátor (v angl. *sparse accumulator*, zkr. SPA). Volba tohoto názvu však může působit poněkud nešťastně, neboť názvy hustý akumulátor a řídký akumulátor



■ **Obrázek 7.9** Ukázka použití ESC akumulátoru. Překresleno z [45].

tor mohou přirozeně vyvolávat dojem, že představují rozdílné datové struktury, nicméně ve vědeckých člancích obě tato označení představují tentýž akumulátor. [1, 53, 59, 60]

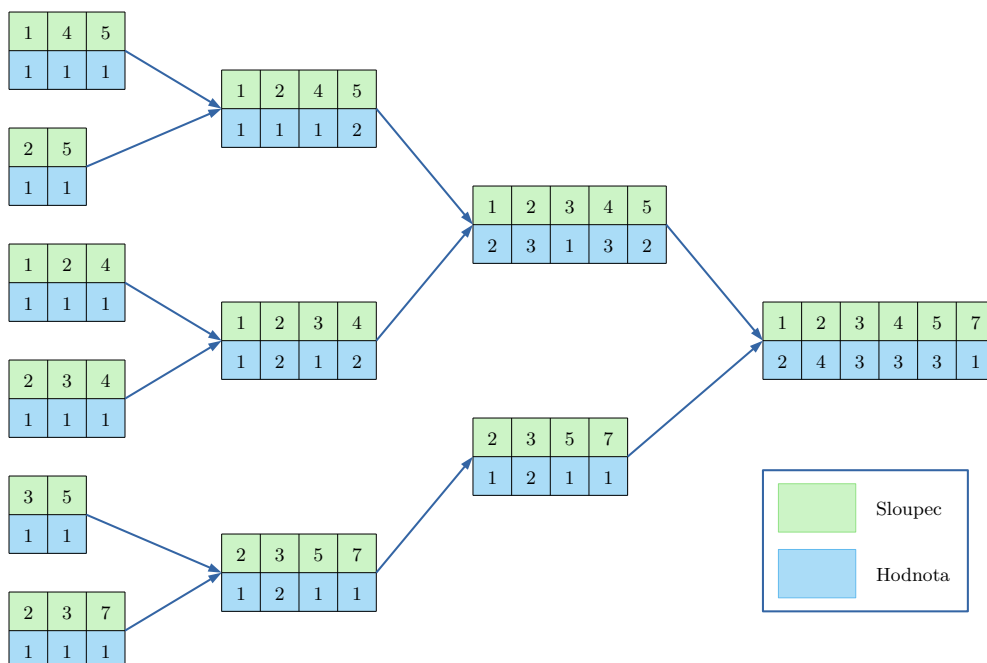
Nejvýznamnější výhodou SPA akumulátoru je přístup k ukládaným prvkům v konstantním čase $O(1)$. Další výhodou je, že tento akumulátor nevyžaduje žádné dodatečné řazení uložených prvků, neboť řazení nahrazuje jedním průchodem přes pole příznaků. Tyto výhody jsou však vyváženy velmi vysokými paměťovými nároky pro matice větších rozměrů a neefektivní práci s pamětí pro řádky, které mají nízký poměr nenulových prvků k celkovému počtu sloupců. [1, 60]

7.3.2 ESC

Alternativní strategií oproti uchovávání mezivýsledků v hustém poli je ukládání dílčích výsledků v kompaktní datové struktuře. Tento způsob využívá akumulátor ESC, který navíc odděluje fázi generování mezivýsledků od jejich akumulace. ESC akumulátor používá neseřazené pole pro uložení mezivýsledku z každého vynásobení dvou nenulových prvků vstupních matic. Společně s mezivýsledkem je ukládán i sloupcový index, na který má být tento mezivýsledek akumulován v řádku výstupní matice. [1, 46]

Ukázku použití ESC akumulátoru zachycuje obrázek 7.9. ESC akumulátor rozděluje správu mezivýsledků do tří samostatných fází:

1. *Expand*: V této fázi jsou do pole uloženy všechny dílčí mezivýsledky ve formě páru hodnoty a sloupcového indexu. Některé implementace kromě sloupcového indexu ukládají navíc i řádkový index. [45, 46]
2. *Sort*: Tato fáze seřadí podle sloupcového indexu všechny záznamy uložené v neseřazeném poli. Článek [46], který akumulátor ESC představil, pro řazení používá algoritmus *radix sort*.



■ **Obrázek 7.10** Ukázka akumulátoru se sléváním. Překresleno z [48].

3. *Compress*: Tato fáze, někdy nazývána také *Contract*, průchodem přes seřazené pole uložených prvků sečte hodnoty, které mají stejný sloupcový index. [45, 46]

Výhodou ESC akumulátoru je, že eliminuje potřebu hustého pole, čímž významně šetří nároky na paměť pro matice velkých rozměrů. Nicméně paměťové nároky stále mohou být poměrně vysoké pro matice, pro které bude v numerické fázi generován vysoký počet mezivýsledků. Další nevýhodou tohoto akumulátoru je samotné řazení, které v případě ESC akumulátoru může dosáhnout až 55 % času celé numerické fáze výpočtu. [1, 45, 46]

7.3.3 Akumulátor založený na slévání

Akumulátor založený na slévání (v angl. *merge-based accumulator*) udržuje několik seřazených posloupností, které po skončení numerické fáze výpočtu slučuje pomocí techniky slévání obdobně, jako to dělá řadící algoritmus *merge sort*. Na rozdíl od předchozích akumulátorů vyžaduje akumulátor se sléváním, aby nenulové prvky v jednotlivých řádcích matice $\mathbf{B}$ byly seřazené podle svého sloupcového indexu. Pokud nejsou nenulové prvky v řádcích matice $\mathbf{B}$ seřazené, je nezbytné každý řádek seřadit ještě před zahájením numerického výpočtu. Ukázka tohoto akumulátoru je k vidění na obrázku 7.10. [1, 48, 49]

Každý řádek výstupní matice $\mathbf{C}_{i*}$ je vypočítán tak, že jsou nejprve vynásobeny nenulovými prvky z řádku $\mathbf{A}_{i*}$ odpovídající řádky matice $\mathbf{B}$. Tímto

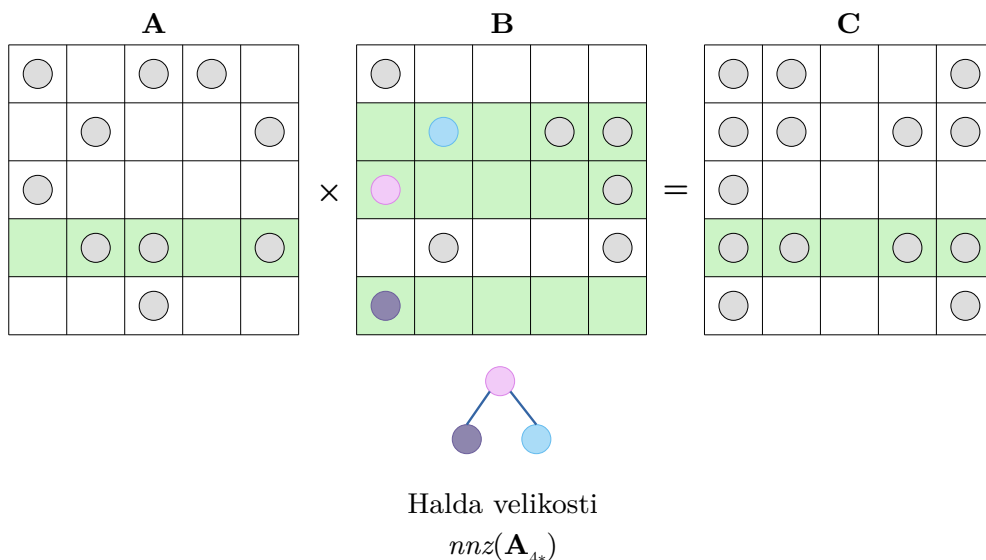
krokem vzniknou seznamy s nenulovými prvky, které jsou již implicitně seřazené podle svého sloupcového indexu. Následně jsou tyto seřazené seznamy hierarchicky slučovány pomocí slévání, kdy se pokaždé sloučí dva seznamy porovnáním sloupcových indexů prvků na začátku obou seznamů, jako je tomu u algoritmu *merge sort*, ovšem s tím rozdílem, že hodnoty se stejnými sloupcovými indexy se sloučí do jednoho prvku. Tímto způsobem zůstávají seznamy seřazené po celou dobu výpočtu daného výstupního řádku. Na každé úrovni se počet řádků ke sloučení sníží přibližně o polovinu, avšak délka těchto řádků narůstá kvůli slučování. [1, 48, 49]

Výhodou tohoto akumulátoru je, že na rozdíl od ESC nevyžaduje explicitní řazení po skončení numerické fáze, neboť udržuje seznamy nenulových prvků seřazené. Tento akumulátor se dále vyznačuje tím, že využití paměti je úměrné počtu nenulových prvků v odpovídajících řádcích matice $\mathbf{B}$. Nicméně, tato vlastnost může být velmi problematická pro vstupní matice, které při násobení generují velký počet mezivýsledků, nebo které mají velký počet téměř hustých řádků. Další nevýhodou je, že tento akumulátor vyžaduje, aby prvky v řádcích matice $\mathbf{B}$ byly seřazené podle svého sloupcového indexu, přičemž pokud seřazené nejsou, je nezbytné každý řádek matice $\mathbf{B}$ před zahájením výpočtu seřadit. [48, 49]

7.3.4 Akumulátor s haldou

Způsob výpočtu akumulátoru s haldou (v angl. *heap-based accumulator*) je velmi podobný předchozímu akumulátoru se sléváním. Zatímco akumulátor založený na slévání slučuje vynásobené řádky matice $\mathbf{B}$ hierarchicky po dvojicích, akumulátor s haldou slučuje najednou všechny řádky matice $\mathbf{B}$, které jsou referencovány nenulovými prvky v řádku matice $\mathbf{A}$. Prvky jsou akumulovány prostřednictvím minimové haldy, která jako klíč pro haldové uspořádání používá sloupcový index uložených prvků. Tento akumulátor stejně jako akumulátor se sléváním vyžaduje, aby nenulové prvky v řádcích matice $\mathbf{B}$ byly seřazené podle svého sloupcového indexu, přičemž pokud seřazené nejsou, je nezbytné tyto prvky seřadit ještě před zahájením numerického násobení. Princip použití tohoto akumulátoru ukazuje obrázek 7.11. [1, 60]

Jelikož všechny řádky matice $\mathbf{B}$ tvoří již seřazené seznamy podle sloupcových indexů nenulových prvků, uchovává halda pouze první prvky všech těchto seznamů. Pro zkonstruování řádku výstupní matice $\mathbf{C}_{i*}$ je tedy použita halda o $nnz(\mathbf{A}_{i*})$ prvcích. Pro každý nenulový prvek $\mathbf{A}_{ik}$ je do haldy vložen první nenulový prvek řádku $\mathbf{B}_{k*}$ spolu s jeho sloupcovým indexem. V každé iteraci akumulátor vybere z haldy prvek s nejmenším sloupcovým indexem, vynásobí jej odpovídajícím prvkem z i -tého řádku matice $\mathbf{A}$ a akumuluje tento mezivýsledek na aktuální pozici ve výstupním řádku nebo jej uloží na novou pozici výstupního řádku, na základě hodnoty sloupcových indexů mezivýsledku a aktuálního prvku ve výstupním řádku $\mathbf{C}_{i*}$. Po této operaci akumulátor vloží do haldy následující nenulový prvek z takového řádku matice $\mathbf{B}$, ze kterého pochá-



■ **Obrázek 7.11** Ukázka akumulátoru s haldou. Inspirováno ilustrací z [54].

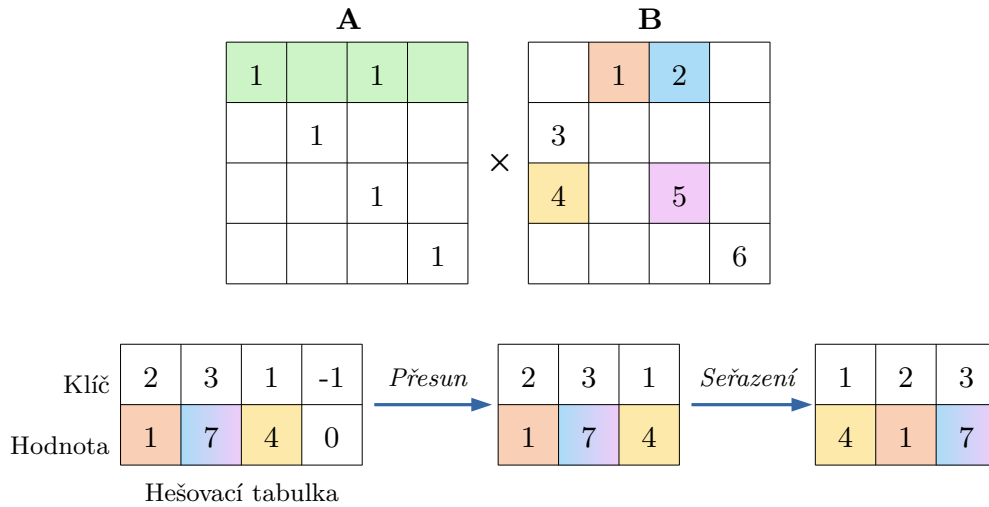
zel poslední prvek extrahovaný z haldy. V okamžiku, kdy se halda vyprázdní, je zkonstruován celý výstupní řádek $\mathbf{C}_{i*}$. [47, 48, 60]

Nejzásadnější výhodou tohoto akumulátoru jsou jeho velmi nízké paměťové nároky, které vůbec nezávisí na struktuře řídkosti matice $\mathbf{B}$, neboť pro zkonstruování jednoho řádku výstupní matice $\mathbf{C}_{i*}$ je zapotřebí pouze $nnz(\mathbf{A}_{i*})$ paměti. Nevýhodou pak je časová náročnost operací s haldou, neboť každé vložení a odebrání prvku z haldy má časovou složitost $O(\log(l))$, kde l označuje počet prvků uložených v haldě. Další nevýhodou je, že stejně jako akumulátor založený na slévání, vyžaduje i akumulátor s haldou, aby prvky v jednotlivých řádcích matice $\mathbf{B}$ byly seřazené podle sloupcového indexu, a pokud seřazené nejsou, je nezbytné nenulové prvky v každém řádku před numerickým násobením seřadit. [47, 48, 60]

7.3.5 Akumulátor s hešováním

Akumulátor s hešováním (v angl. *hash-based accumulator*) používá pro akumulaci nenulových prvků hešovací tabulku. Tento akumulátor pro alokaci paměťového prostoru hešovací tabulky typicky používá metodu horního odhadu, přičemž jako klíč používá sloupcový index nenulových prvků. Příklad použití akumulátoru s hešováním ukazuje obrázek 7.12. [1, 45, 60]

Vzhledem k tomu, že sloupcové indexy jsou vždy nezáporné, je možné na začátku výpočtu hešovací tabulku inicializovat zápornou hodnotou, například -1 . Jakmile je akumulace a výpočet výstupních nenulových prvků pomocí hešovací tabulky dokončen, je potřeba uložené prvky z hešovací tabulky přesunout do paměťového prostoru, který je vyhrazen pro daný řádek výstupní



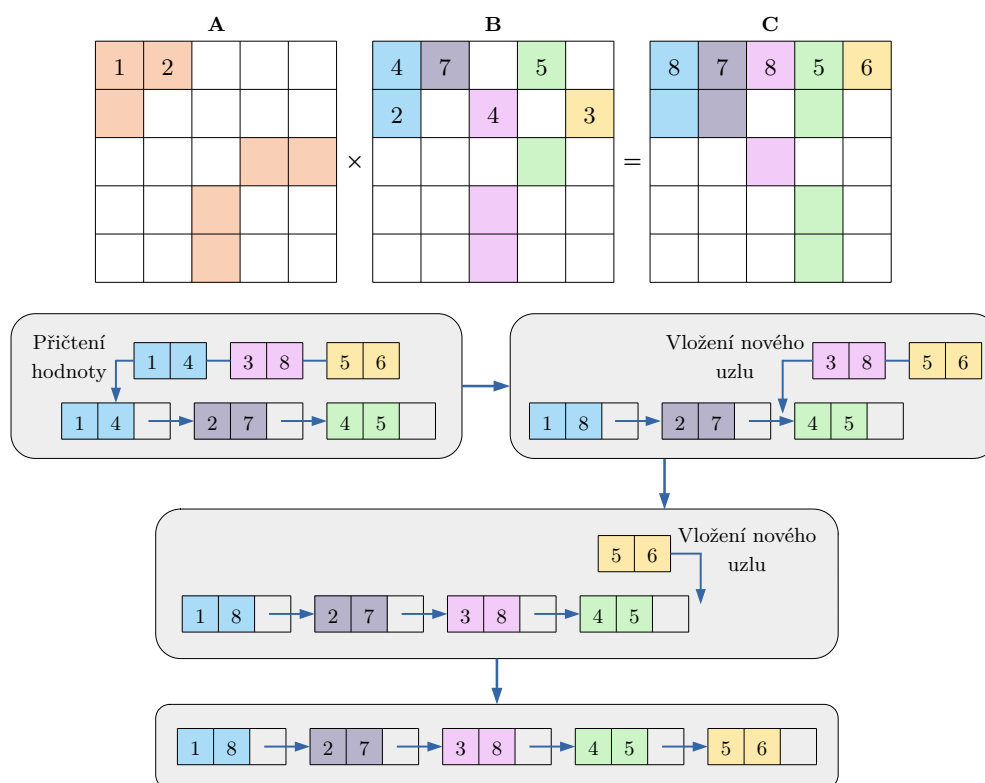
■ **Obrázek 7.12** Ukázka akumulátoru s hešováním. Inspirováno ilustrací z [1].

matice **C**. Následně jsou tyto nenulové prvky seřazeny podle jejich sloupcového indexu. Nicméně seřazení nenulových prvků může být vynecháno, pokud je povoleno, aby nenulové prvky výstupní matice byly v rámci jednoho řádku neseřazené, což povoluje např. reprezentace v CSR formátu (viz. kapitola 6). Tím se řazení použité v akumulátoru s hešovací tabulkou zásadně liší od ESC, kde je seřazení nezbytné k tomu, aby následně mohly být akumulovány mezivýsledky se stejným sloupcovým indexem. Jak je popsáno ve článku [60], vynecháním řazení v akumulátoru s hešovací tabulkou lze dosáhnout výrazného zrychlení celého algoritmu. [1, 45, 60]

Nejvýznamnější výhodou akumulátoru s hešováním je, že poskytuje amortizovaně konstantní složitost $O^*(1)$ přístupu k ukládaným prvkům bez alokace hustého pole, jako je tomu v případě SPA akumulátoru. Díky této vlastnosti je akumulátor s hešováním vhodnou volbou pro řídké matice velkých rozměrů, pro které je SPA akumulátor paměťově neefektivní. Další výhodou je, že na rozdíl od akumulátorů se sléváním a s haldou tento akumulátor nevyžaduje seřazené prvky v řádcích matice **B**. Nevýhodou jsou hešovací konflikty, které mohou snížit výkon celého akumulátoru. Ke snížení výkonu také přispívá řazení nenulových prvků po skončení numerického násobení, nicméně výhodou je, že na rozdíl od ESC může tento akumulátor řazení zcela vynechat, pokud není uživatelem explicitně vyžadováno. [1, 45, 60]

7.3.6 LLA

Akumulátor založený na spojovém seznamu (z angl. *linked list-based accumulator*, zkr. LLA) eliminuje explicitní řazení výstupních nenulových prvků tím, že po celou dobu akumulace jednoho řádku matice **C** uchovává ve spojovém seznamu nenulové prvky v seřazeném stavu. Avšak stejně jako akumulátor za-

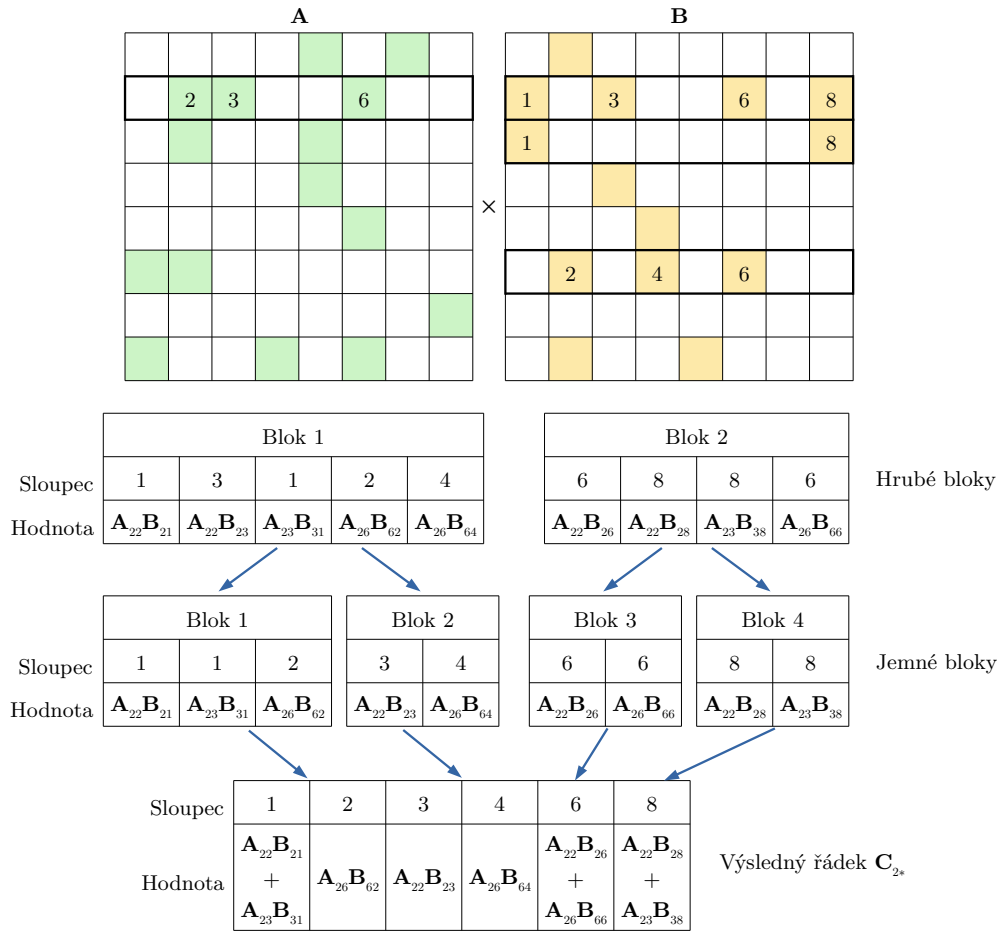


■ **Obrázek 7.13** Ukázka použití LLA akumulátoru. Překresleno z [45].

ložený na slévání a akumulátor s haldou vyžaduje i LLA, aby nenulové prvky matice **B** byly seřazené podle svých sloupcových indexů. Ukázka použití LLA akumulátoru je k vidění na obrázku 7.13. [45]

Každý uzel spojového seznamu obsahuje tři položky. Položka *key* uchovává sloupcový index daného nenulového prvku, položka *value* uchovává jeho hodnotu a položka *pos* uchovává adresu následujícího uzlu spojového seznamu. Řádek výstupní matice **C** je vypočítán sloučením řádků matice **B**, jejichž nenulové prvky jsou vynásobeny hodnotami prvků z odpovídajícího řádku matice **A**. LLA akumulátor při výpočtu vždy přistupuje k celému řádku matice **B**, který po vynásobení odpovídajícím nenulovým prvkem z řádku matice **A** následně vloží prvek po prvků do spojového seznamu. Po dokončení akumulace všech odpovídajících řádků matice **B** jsou prvky ze spojového seznamu následně přesunuty do paměťového prostoru, který je určen pro daný výstupní řádek matice **C**. [45]

Je zřejmé, že největší výhodou tohoto akumulátoru je, že zcela eliminuje řazení nenulových prvků výstupního řádku matice **C**, neboť nenulové prvky udržuje v průběhu výpočtu seřazené. Avšak tato výhoda je vyvážena tím, že je nutné před zahájením numerického násobení seřadit nenulové prvky v každém řádku matice **B**, obdobně jako u akumulátorů se sléváním a s haldou.



■ **Obrázek 7.14** Ukázka výpočtu algoritmu MAGNUS. Překresleno z [5].

Další nevýhodou je, že při každém vkládání všech nenulových prvků z jednoho řádku matice **B** do spojového seznamu akumulátoru může být potřeba projít téměř celý seznam nebo jeho značnou část, což může pro některé matice vést k významné degradaci výkonu algoritmu. [45]

7.3.7 MAGNUS

Algoritmus MAGNUS (zkr. pro *Matrix Algebra for Gigantic Numerical Systems*) se liší od předchozích přístupů tím, že SpGEMM násobení dekomponuje na menší dílčí výpočty, které je možné provádět v rámci cache paměti procesoru. Zatímco ostatní akumulátory představené v této kapitole se soustředí na optimalizaci své datové struktury a celého procesu akumulace, algoritmus MAGNUS se zaměřuje na optimalizaci vzorců přístupu k datům. Klíčovou myšlenkou je přeuspořádání dílčích násobení nenulových prvků do bloků, jejichž velikost nepřekročí kapacitu L2 cache paměti procesoru (viz. kapitola 2.2). [5]

Pro vygenerování těchto bloků je aplikován dvouúrovňový hierarchický přístup. Hrubý algoritmus (v angl. *coarse-level algorithm*) je založen na CbR formulaci násobení matic (viz. kapitola 5) a představuje první úroveň dekompozice dat. Jemný algoritmus (v angl. *fine-level algorithm*) je založen na RbR formulaci násobení matic (viz. kapitola 5) a dále přeskupuje a dekomponuje bloky, které byly vypočítány hrubým algoritmem. Počet bloků použitých v obou algoritmech je vypočítán pomocí vlastností vstupních matic a specifikací cílového výpočetního systému. Následně je pro každý blok, který byl vypočítán jemným algoritmem, zvolen jeden ze dvou akumulátorů. Prvním akumulátorem je ESC akumulátor (viz. 7.3.2), který pro řazení využívá specializovanou variantu Quicksort algoritmu, jenž je vektorizovaný procesorovými instrukcemi AVX-512. Tento akumulátor je zvolen pro bloky o 256 prvcích a méně, pro ostatní bloky se používá SPA akumulátor (viz. kapitola 7.3.1). Ukázkou výpočtu dvou řádků výstupní matice $\mathbf{C}$ prostřednictvím algoritmu MAGNUS ilustruje obrázek 7.14. [5, 61]

Algoritmus MAGNUS na maticích zvolených ve článku [5] ve většině případů poráží všechny ostatní metody. Nevýhodou tohoto algoritmu však je špatná přenositelnost, která plyne z jeho silné hardwarové závislosti na instrukční sadě AVX-512. Tato sada je kromě vektorizace řazení v akumulátoru ESC použita také pro instrukce typu `_mm512_stream_si512()`, které umožňují zápis dat do hlavní paměti, aniž by tento zápis vyřadil některá existující data v cache paměti procesoru. Další nevýhodou je velmi vysoká složitost implementace, která může vést ke špatné udržitelnosti, rozšiřitelnosti a znovupoužití implementovaných funkcionalit. [5]



Kapitola 8

Návrh

Tato kapitola se zabývá návrhem algoritmu paralelního násobení řídkých matic a jeho pomocných struktur. Nejprve je navržen a popsán obecný algoritmus pro násobení řídkých matic včetně dílčích částí tohoto algoritmu. Dále je navržena hešovací tabulka pro akumulátor s hešováním včetně analýzy a návrhu vhodné hešovací funkce. Na závěr je navržena zásobárna vláken určená pro paralelizaci SpGEMM algoritmu v C++20 bez externích knihoven pro paralelizaci.

Návrh algoritmu pro paralelní násobení řídkých matic bude vycházet z přístupů prezentovaných ve článku [60], který se věnuje optimalizaci SpGEMM problému pro systémy se sdílenou pamětí s procesory Intel Xeon a Intel Xeon Phi, což jsou představitelé vícejádrových a mnohójádrových procesorů. Autory prezentované řešení navíc na většině testovaných dat dosahuje lepších výsledků než implementace SpGEMM v knihovnách Intel MKL a Kokkos Kernels. Z těchto důvodů jsou přístupy prezentované v tomto článku ideální volbou pro návrh aplikací určených k porovnání a vyhodnocení paralelizačních přístupů pro problém násobení řídkých matic na výpočetních architekturách se sdílenou pamětí.

Ve článku [60] je porovnáván výkon akumulátoru s haldou a akumulátoru s hešováním. Akumulátor s hašováním dosahoval výrazně lepších výsledků na většině testovaných dat, zatímco akumulátor s haldou dosahoval lepších výsledků pouze na velmi specifických maticích, které mají nízký poměr počtu operací násobení k počtu nenulových prvků. Významným zjištěním článku [60] pak je, že SpGEMM založený na akumulátoru s hešováním dosahuje výrazného zvýšení výkonu, pokud není vyžadováno, aby byly nenulové prvky každého řádku výstupní matice seřazené podle sloupcového indexu. Tato varianta akumulátoru s hešováním dosahovala v průměru zrychlení 1.68 na maticích z kolekce SuiteSparse Matrix Collection [24], než akumulátor s hešováním, který nenulové prvky v každém řádku výstupní matice seřazuje. Protože akumulátor s hešováním navíc nevyžaduje, aby byly nenulové prvky v jednotlivých řádcích

vstupních seřazené, byl z těchto důvodů pro účely této práce jako akumulátor zvolen akumulátor s hešováním. Konkrétnímu návrhu hešovací tabulky se věnuje kapitola 8.2.

Pro násobení řídkých matic používají autoři článku [60] RbR formulaci, přičemž vstupní i výstupní matice jsou reprezentovány ve formátu komprimovaných řádků CSR. Pro predikci počtu nenulových prvků výstupní matice autoři využívají metodu přesné alokace a pro rozdělení výpočetní zátěže je využito jednorozměrné T -rozdělení, přičemž pro přiřazení výpočtu konkrétních řádků výstupní matice jednotlivým vláknům je použita predikce počtu operací násobení nenulových prvků v dílčích řádcích. Všechny tyto přístupy budou adaptovány i pro účely návrhu aplikací v této práci.

8.1 Algoritmus násobení řídkých matic

Pseudokód navrhovaného řešení ukazuje algoritmus 8.1. Nejprve je vypočítán požadovaný počet operací násobení nenulových prvků pro každý řádek výstupní matice $\mathbf{C}$. Protože tento výsledek reprezentuje náročnost výpočtu jednotlivých řádků výstupní matice, je následně použit k rozdělení zpracování řádků matice $\mathbf{C}$ příslušným výpočetním vláknům. V dalším kroku je predikce počtu operací násobení využita pro alokaci paměti v hešovacích tabulkách jednotlivých vláken. Následně je spuštěna symbolická fáze, která vypočítá počet nenulových prvků výstupní matice pomocí metody přesné alokace a na základě tohoto výsledku nastaví pole ukazatelů matice $\mathbf{C}$ a zároveň alokuje paměť pro pole sloupců a pole hodnot. Na závěr je spuštěna numerická fáze, která vypočítává konkrétní hodnoty nenulových prvků výstupní matice.

Vstup : Řídké matice $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$ v CSR formátu

Výstup: Řídká matice $\mathbf{C} \in \mathbb{R}^{m,p}$ v CSR formátu

```

1  $flops \leftarrow \text{PredictFlops}(\mathbf{A}, \mathbf{B})$ 
2  $offsets \leftarrow \text{RowsToThreads}(flops, m)$ 
3  $tables \leftarrow \text{AllocateHashTables}(flops, offsets, p)$ 
4  $rpts_{\mathbf{C}} \leftarrow \text{SymbolicPhase}(\mathbf{A}, \mathbf{B}, offsets, tables)$ 
5  $\mathbf{C} \leftarrow \text{NumericPhase}(\mathbf{A}, \mathbf{B}, rpts_{\mathbf{C}}, offsets, tables)$ 
6 return  $\mathbf{C}$ 
    
```

Algoritmus 8.1: Pseudokód algoritmu Hash SpGEMM. Inspirováno pseudokódem z [60].

Pseudokód výpočtu nezbytného počtu operací násobení nenulových prvků v každém řádku výstupní matice $\mathbf{C}$ ukazuje algoritmus 8.2. Je zřejmé, že tento algoritmus je velmi podobný metodě horního odhadu počtu nenulových prvků (viz. 7.1.1). Avšak tento algoritmus na rozdíl od metody horního odhadu nezastrěšuje predikci počtem sloupců výstupní matice, tj. výsledná predikce může být vyšší, než je počet sloupců matice $\mathbf{C}$. V pseudokódu ve článku [60] je pro rozsah hodnot **for** cyklů nepřesně použito klíčové slovo **to**, které se používá

spíše pro uzavřený rozsah hodnot. Z toho důvodu bylo pro lepší zřetelnost nahrazeno klíčovým slovem *until*. Formulace $k \text{ until } m$ tedy reprezentuje rozsah celočíselných hodnot $k, k + 1, \dots, m - 1$.

Vstup : Řídké matice $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$ v CSR formátu

Výstup: Pole *flops*

```

1 for  $i \leftarrow 0$  until  $m$  in parallel do
2    $flops[i] \leftarrow 0$ 
3   for  $j \leftarrow rpts_{\mathbf{A}}[i]$  until  $rpts_{\mathbf{A}}[i + 1]$  do
4      $rnnz \leftarrow rpts_{\mathbf{B}}[cols_{\mathbf{A}}[j] + 1] - rpts_{\mathbf{B}}[cols_{\mathbf{A}}[j]]$ 
5      $flops[i] \leftarrow flops[i] + rnnz$ 
6   end
7 end
8 return flops
    
```

Algoritmus 8.2: Pseudokód algoritmu PredictFlops. S úpravami převzato ze článku [60].

Pseudokód rozdělení zpracování řádků výstupní matice $\mathbf{C}$ mezi dostupná vlákna tak, aby bylo výpočetní zatížení rozděleno co nejrovnoměrněji, ukazuje algoritmus 8.3. Vstupem je pole *flops*, jehož i -tý prvek udává počet operací násobení nenulových prvků potřebných k výpočtu i -tého řádku výstupní matice a dále počet řádků m výstupní matice. Výstupem je pole *offsets*, které pro každé vlákno udává index prvního řádku, který má toto vlákno zpracovat, přičemž vlákno s identifikátorem *tid* bude zpracovávat řádky z rozsahu $\langle offsets[tid]; offsets[tid + 1] \rangle$.

Algoritmus iterativně určuje rozsahy výpočtu řádků výstupní matice pro jednotlivá vlákna. Počáteční index aktuálního vlákna zároveň určuje výpočetní zátěž předchozího vlákna. Nejprve je proto vypočítána cílová výpočetní zátěž předchozího vlákna $target_{flops}$ jako podíl zbývajících operací násobení a zbývajících počtu vláken. Následně je vypočítán cílový kumulativní počet operací násobení $target_{ps}$ jako součet již přiřazených operací násobení a $target_{flops}$. Počáteční index aktuálního vlákna je pak určen jako index první hodnoty v poli *flops_{ps}*, jejíž hodnota je větší než $target_{ps}$.

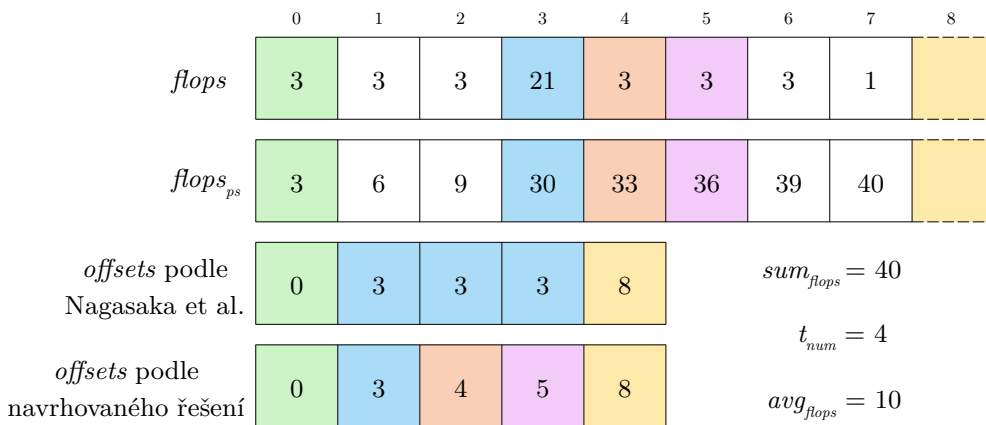
Původní algoritmus RowsToThreads ze článku [60] vypočítává počáteční index vlákna jako první prvek v poli prefixových součtů *flops_{ps}*, který je větší nebo roven hodnotě $avg_{flops} \cdot tid$, kde avg_{flops} představuje průměrnou zátěž každého vlákna. Tento přístup však může vést k velmi nerovnoměrnému rozdělení výpočetní zátěže. Pokud jsou například k dispozici 4 vlákna, celkový počet operací násobení je 40, průměrný cílový počet operací na vlákno je 10 a pole prefixových součtů *flops_{ps}* obsahuje hodnoty, které jsou zobrazeny na obrázku 8.1, pak první vlákno obdrží 9 operací násobení, druhé i třetí vlákno žádné a poslední vlákno obdrží 31 operací násobení, což je velmi nevyvážené rozdělení. Naopak řešení navrhané v této práci rozděluje výpočetní zátěž mnohem rovnoměrněji, jak je také vidět na obrázku 8.1.

Vstup : Pole $flops$, počet řádků m výstupní matice $\mathbf{C} \in \mathbb{R}^{m,p}$
Výstup: Pole $offsets$

```

1  $flops_{ps} \leftarrow \text{ParallelPrefixSum}(flops)$ 
2  $t_{num} \leftarrow \text{GetThreadCount}()$ 
3  $t_{left} \leftarrow t_{num}$ 
4  $offsets[0] \leftarrow 0$ 
5 for  $tid \leftarrow 1$  until  $t_{num}$  do
6      $prev \leftarrow offsets[tid - 1]$ 
7     if  $prev = m$  then
8          $offsets[tid] \leftarrow m$ 
9     else
10         $assigned_{flops} \leftarrow 0$ 
11        if  $prev > 0$  then
12             $assigned_{flops} \leftarrow flops_{ps}[prev - 1]$ 
13        end
14         $remaining_{flops} \leftarrow flops_{ps}[m - 1] - assigned_{flops}$ 
15         $target_{flops} \leftarrow \text{Round}(remaining_{flops} / t_{left})$ 
16         $target_{ps} \leftarrow assigned_{flops} + target_{flops}$ 
17        if  $flops[prev] \geq target_{flops}$  then
18             $offsets[tid] \leftarrow prev + 1$ 
19        else
20             $offsets[tid] \leftarrow \text{UpperBound}(flops_{ps}, target_{ps})$ 
21        end
22    end
23     $t_{left} \leftarrow t_{left} - 1$ 
24 end
25  $offsets[t_{num}] \leftarrow m$ 
26 return  $offsets$ 
    
```

Algoritmus 8.3: Pseudokód algoritmu RowsToThreads.



Obrázek 8.1 Ukázka výpočtu pole $offsets$ podle [60] a podle řešení navrhovaného v kapitole 8.1.

Výhodou původního řešení algoritmu RowsToThreads ze článku [60] je, že jej lze paralelizovat, což u řešení navrhovaného v této práci kvůli datové závislosti z předchozích iterací není možné. Nicméně vzhledem k tomu, že počet vláken na systémech se sdílenou pamětí obvykle dosahuje vyšších desítek nebo nižších stovek vláken, nemá absence paralelizace tohoto výpočtu významný dopad. Naopak je mnohem důležitější zajistit, aby výpočetní zátěž zpracování řádků výstupní matice byla mezi jednotlivá vlákna rozdělena co nejrovnoměrněji. Z těchto důvodů proto bylo nakonec paralelní řešení ze článku [60] zavrženo a místo tohoto přístupu bylo navrženo a zvoleno řešení, které je popsáno výše a jež ukazuje algoritmus 8.3.

Vstup : Pole *flops*, pole *offsets*, počet sloupců p výstupní matice
 $\mathbf{C} \in \mathbb{R}^{m,p}$

Výstup: Pole *tables*

```

1  $t_{num} \leftarrow \text{GetMaxThreads}()$ 
2 for  $t_{id} \leftarrow 0$  until  $t_{num}$  in parallel do
3    $size_t \leftarrow 0$ 
4   for  $i \leftarrow offsets[t_{id}]$  until  $offsets[t_{id} + 1]$  do
5      $size_t \leftarrow \text{Max}(size_t, flops[i])$ 
6   end
7    $size_t \leftarrow \text{Min}(size_t, p)$ 
8    $size_t \leftarrow \text{LowestP2}(size_t)$ 
9    $\text{ReserveSpace}(tables[t_{id}], size_t)$ 
10 end
11 return tables
    
```

Algoritmus 8.4: Pseudokód algoritmu AllocateHashTables.

S úpravami převzato ze článku [60].

Pseudokód alokace hešovacích tabulek pro jednotlivá výpočetní vlákna ukazuje algoritmus 8.4. Účelem tohoto algoritmu je vypočítat horní limit velikosti hešovací tabulky každého vlákna a rezervovat paměťový prostor dané hešovací tabulky na základě tohoto horního limitu, aby každé vlákno mohlo tabulku opakovaně používat během symbolické a numerické fáze tím, že ji pro každý řádek znovu inicializuje. Díky tomuto kroku je zaručeno, že žádné vlákno nebude muset během symbolické či numerické fáze tabulku přehesovat, neboť tabulka bude mít vždy dostatečnou kapacitu na akumulaci všech nenulových prvků z daného řádku výstupní matice. Horní limit velikosti hešovací tabulky každého vlákna se určuje jako maximální počet operací násobení na řádek v rámci řádků přiřazených danému vláknu. Tento horní limit je z důvodu efektivního výpočtu operace modulo (viz. 8.2) poté zaokrouhlen na číslo 2^n takové, aby platilo $2^n > size_t$.

Účelem symbolické fáze je výpočet pole ukazatelů výstupní matice $\mathbf{C}$ v CSR formátu. V symbolické fázi jsou tedy do hešovací tabulky vkládány pouze indexy nenulových prvků z daného řádku matice $\mathbf{C}$ bez provedení samotného

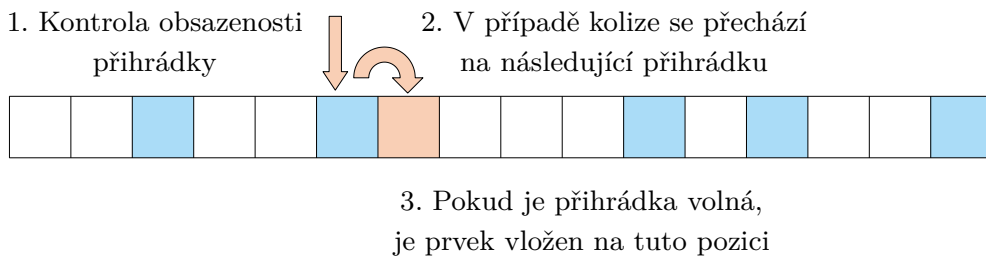
Vstup : Řídké matice $\mathbf{A} \in \mathbb{R}^{m,n}$ a $\mathbf{B} \in \mathbb{R}^{n,p}$ v CSR formátu, pole ukazatelů $rpts_{\mathbf{C}}$ matice $\mathbf{C} \in \mathbb{R}^{m,p}$, pole $offsets$ a pole $tables$

Výstup: Řídká matice $\mathbf{C} \in \mathbb{R}^{m,p}$ v CSR formátu

```

1   $t_{num} \leftarrow \text{GetMaxThreads}()$ 
2  for  $tid \leftarrow 0$  until  $t_{num}$  in parallel do
3      for  $i \leftarrow offsets[tid]$  until  $offsets[tid + 1]$  do
4           $rstart_{\mathbf{A}} \leftarrow rpts_{\mathbf{A}}[i]$ 
5           $rend_{\mathbf{A}} \leftarrow rpts_{\mathbf{A}}[i + 1]$ 
6          for  $idx_{\mathbf{A}} \leftarrow rstart_{\mathbf{A}}$  until  $rend_{\mathbf{A}}$  do
7               $j \leftarrow cols_{\mathbf{A}}[idx_{\mathbf{A}}]$ 
8               $val_{\mathbf{A}} \leftarrow vals_{\mathbf{A}}[idx_{\mathbf{A}}]$ 
9               $rstart_{\mathbf{B}} \leftarrow rpts_{\mathbf{B}}[j]$ 
10              $rend_{\mathbf{B}} \leftarrow rpts_{\mathbf{B}}[j + 1]$ 
11             for  $idx_{\mathbf{B}} \leftarrow rstart_{\mathbf{B}}$  until  $rend_{\mathbf{B}}$  do
12                  $k \leftarrow cols_{\mathbf{B}}[idx_{\mathbf{B}}]$ 
13                  $val_{\mathbf{B}} \leftarrow vals_{\mathbf{B}}[idx_{\mathbf{B}}]$ 
14                  $tables[tid][k] \leftarrow tables[tid][k] + val_{\mathbf{A}} \cdot val_{\mathbf{B}}$ 
15             end
16         end
17          $idx_{out} \leftarrow rpts_{\mathbf{C}}[i]$ 
18         foreach  $[col, val]$  in  $tables[tid]$  do
19              $cols_{\mathbf{C}}[idx_{out}] \leftarrow col$ 
20              $vals_{\mathbf{C}}[idx_{out}] \leftarrow val$ 
21              $idx_{out} \leftarrow idx_{out} + 1$ 
22         end
23     end
24 end
25 return  $(rpts_{\mathbf{C}}, cols_{\mathbf{C}}, vals_{\mathbf{C}})$ 
    
```

Algoritmus 8.5: Pseudokód algoritmu NumericPhase.



Obrázek 8.2 Ukázka hešovací tabulky s otevřeným adresováním a lineárním přidáváním. Překresleno z [60].

násobení. Na konci zpracování každého řádku tak počet prvků v tabulce udává počet nenulových prvků daného řádku. Tento údaj je zapsán na pozici odpovídajícího řádku do pomocného pole a po zpracování všech řádků je pomocí prefixového součtu zkonstruováno výsledné pole ukazatelů. Vzhledem k tomu, že je symbolická fáze velmi podobná numerické fázi, není její pseudokód zde uváděn.

Numerická fáze se od symbolické liší tím, že do hešovací tabulky již akumuluje součin nenulových prvků daného řádku matice **C** a po dokončení akumulace pak nenulové prvky z hešovací tabulky společně s jejich indexy zapisuje do pole prvků a pole sloupcových indexů výstupní matice. Pseudokód numerické fáze ukazuje algoritmus 8.5.

8.2 Hešovací tabulka

Existují dvě kategorie hešovacích tabulek: hešovací tabulka s řetězením a hešovací tabulka s otevřenou adresací. Hešovací tabulka s řetězením řeší kolize, tj. situace, kdy mají několik prvků stejnou heš, tím, že neukládá do přihrádek tabulky prvky samotné, ale ukládá do tabulky pouze odkazy na spojové seznamy, ve kterých jsou uloženy prvky, jejichž klíč má stejnou heš. Naproti tomu hešovací tabulka s otevřenou adresací ukládá prvky přímo do přihrádek tabulky a kolize řeší tím, že je každému prvku přiřazena vyhledávací posloupnost, jež určuje pořadí přihrádek, do kterých se má prvek vložit, přičemž pro uložení daného prvku se použije první volná přihrádka. [18]

Ve článku [60] je pro akumulátor použita hešovací tabulka s otevřenou adresací a lineárním přidáváním jako metodou určování vyhledávací posloupnosti. Ačkoliv nejsou ve článku uvedeny konkrétní důvody volby této hešovací tabulky, je zřejmé, že jedním z hlavních důvodů jsou její dobré vlastnosti paměťové prostorové lokality (viz. 2.2). Položky spojového seznamu hešovací tabulky s řetězením většinou bývají od sebe v paměťovém prostoru vzdálené, neboť paměťový prostor každé nové položky spojového seznamu je alokován na haldě (viz. 2.4) separátně od ostatních položek. Adresa následující položky spojového seznamu navíc není známá, dokud není v aktuální položce seznamu přistoupeno k ukazateli na následující položku. Přístup ke každé položce spojového seznamu proto velmi pravděpodobně povede k výpadku cache paměti a nahrání potřebných dat z hlavní paměti systému. Hešovací tabulka s otevřenou adresací však ukládá prvky přímo do přihrádek svého souvislého paměťového prostoru. Pokud je pro vyhledávací posloupnost navíc použita metoda lineárního přidávání, je pravděpodobné, že po přístupu k prvnímu klíči bude prvek s dalším klíčem již nahrán v cache paměti procesoru. Mechanismus lineárního přidávání pro hešovací tabulku s otevřenou adresací ukazuje obrázek 8.2. Hešovací tabulka popsaná ve článku [60] také pro efektivní výpočet operace modulo používá kapacitu velikosti 2^n , kde $n \in \mathbb{N}$ je přirozené číslo.

Ze zdrojových kódů plyne, že kompilátory GCC a Clang pro implementaci `std::unordered_map` používají hešovací tabulku s řetězením. Tato volba

je pravděpodobně důsledkem požadavku C++ standardu, který stanovuje, že k přehešování tabulky může docházet pouze při překročení faktoru naplnění tabulky, přičemž výchozí faktor naplnění je 1.0. [62, 63]. Vzhledem k tomu, že efektivita hešovací tabulky s otevřeným adresováním kvůli dlouhým vyhledávacím posloupnostem významně klesá, pokud je faktor naplnění blízký hodnotě 1.0, je přirozené, že kompilátory GCC i Clang používají právě hešovací tabulku s řetězením. Implementace `std::unordered_map` v kompilátoru GCC navíc používá prvočíselnou kapacitu hešovací tabulky místo kapacity 2^n pro efektivní výpočet operace modulo. Z uvedených důvodů je proto zřejmé, že pro účely akumulátoru s hešováním v rámci této práce nelze použít kontejner ze standardní knihovny `std::unordered_map`.

Hešovací tabulka s otevřeným adresováním a kapacitou 2^n je sice použita například v knihovně Boost v kontejneru `boost::unordered_flat_map`, nicméně tento kontejner nepoužívá jako hešovací funkci vyšší bity součinu s multiplikativní konstantou Donalda Knutha (viz. dále). Tato hešovací funkce je vhodnou volbou pro hešovací tabulky s otevřeným adresováním o kapacitě 2^n , avšak protože pro výpočet heše používá interní informaci o aktuální kapacitě tabulky, nelze tuto hešovací funkci použít externě jako parametr pro již existující implementaci hešovací tabulky. Navíc, knihovna Boost není k dispozici na výpočetních ARM uzlech fakultního klastru ClusterFIT, na kterém budou prováděna všechna měření v rámci této práce. Z těchto důvodů plyne, že je nezbytné pro účely této práce implementovat vlastní řešení.

Cílem tedy je navrhnout a posléze implementovat hešovací tabulku s otevřeným adresováním a lineárním přidáváním. Stejně jako ve článku [60] bude tabulka používat kapacitu 2^n pro efektivní výpočet operace modulo prostřednictvím bitového logického součinu. Zbývá tedy navrhnout vhodnou hešovací funkci, která efektivně vypočítá heš vstupního klíče a zároveň každou k -prvkovou množinu vstupních klíčů rozprostře mezi přihrádky tabulky co nejrovnoměrněji. Návrh takové hešovací funkce je však mimořádně náročná úloha, je proto žádoucí zvolit již existující hešovací funkci, která však bude vhodnou volbou pro navrhovanou hešovací tabulku. Z kapitoly o hešování v knize [18] plyne, že ideální volbou je použít vyšší bity součinu. Tato hešovací funkce se používá pro tabulky, které mají 2^n přihrádek, přičemž heš každého w -bitové klíče x se pomocí této hešovací funkce vypočítá následovně:

$$x \mapsto \left\lfloor \frac{ax \bmod 2^w}{2^{w-n}} \right\rfloor.$$

Každý w -bitový klíč x je nejprve vynásoben lichou w -bitovou konstantou a , tento součin je oříznut na w bitů a z výsledku je pro heš použito nejvyšších n bitů. Tato hešovací funkce je tedy vhodnou volbou nejen proto, že je navržena cíleně pro hešovací tabulky s 2^n přihrádkami, ale také proto, že její výpočet je velice rychlý, neboť přetečení ve většině programovacích jazyků automaticky ořezává výsledek a k vyhodnocení funkce tedy stačí pouze jedno násobení a následný bitový posuv. [18]

Kvalita této hešovací funkce však očividně závisí na volbě multiplikativní konstanty a . Problematikou volby vhodné konstanty se zabýval i matematik a informatik Donald Knuth ve své knize [64], ve které navrhuje konstantu odvozenou od převrácené hodnoty zlatého řezu ϕ . Stejný postup je navrhován také v kapitole o hešovacích funkcích v knize [65], kde je jako konstanta a doporučena hodnota $A \cdot 2^w$, kde A je převrácená hodnota zlatého řezu, tj.

$$A = \frac{1}{\phi} = \frac{2}{\sqrt{5} + 1} = \frac{\sqrt{5} - 1}{2}.$$

Pro $w = 64$, tj. pro 64-bitové klíče, je hodnotě $A \cdot 2^w$ podle nástroje Wolfram|Alpha [66] nejbližší číslo `0x9E3779B97F4A7C16`. Pro vyšší bity součinu je však potřebná lichá multiplikativní konstanta, neboť aby operace $ax \bmod 2^w$ byla bijekcí pro všechna w -bitová čísla, musí být konstanta a nesoudělná s modulem 2^w . Vhodnější proto je zvolit jako hodnotu a druhé nejbližší číslo, tj. `0x9E3779B97F4A7C15`.

8.3 Zásobárna vláken

Jedním z cílů této práce je implementace SpGEMM algoritmu v jazyce C++ bez použití externích knihoven pro paralelizaci. Od C++17 je sice možné algoritmy ze standardní knihovny STL spouštět v paralelním režimu, ale jak bylo popsáno v kapitole 3.1.6, specifikováním paralelního způsobu provádění není zaručeno, že výpočet daného algoritmu skutečně bude proveden paralelně. Je možné, že výpočet daného algoritmu bude proveden sekvenčně i přes explicitní specifikaci upřednostnění paralelizace, přičemž rozhraní těchto algoritmů neposkytuje způsob, prostřednictvím kterého by bylo možné zjistit, zda daný algoritmus byl proveden sekvenčně, nebo paralelně a kolik vláken bylo použitých při samotném výpočtu.

Z uvedených důvodů je zřejmé, že tento způsob paralelizace nelze pro vědecké výpočty v rámci této práce použít, neboť by jejich porovnání s výsledky paralelní varianty v OpenMP nemělo vypovídací hodnotu. Je proto nezbytné navrhnout a implementovat vlastní řešení založené na vláknech ze standardní knihovny STL.

Vytváření a ukončování vláken je poměrně nákladná operace, proto může OpenMP pro snížení této režie mezi paralelními oblastmi používat zásobárnu recyklovatelných vláken (z angl. *thread pool*). Aby nebyla znevýhodněna implementace využívající C++ vlákna, bude pro účely této varianty implementována vlastní zásobárna vláken. To je z hlediska implementace, ale i návrhu, velmi náročná úloha, neboť je nutné navrhnout rozhraní této třídy tak, aby umožňovalo jednoduchým způsobem přepsat paralelní konstrukty z OpenMP varianty implementace SpGEMM algoritmu.

Je zřejmé, že zásobárna vláken musí podporovat datový paralelismus prostřednictvím paralelizace smyčky `for`. Zároveň také musí podporovat různé

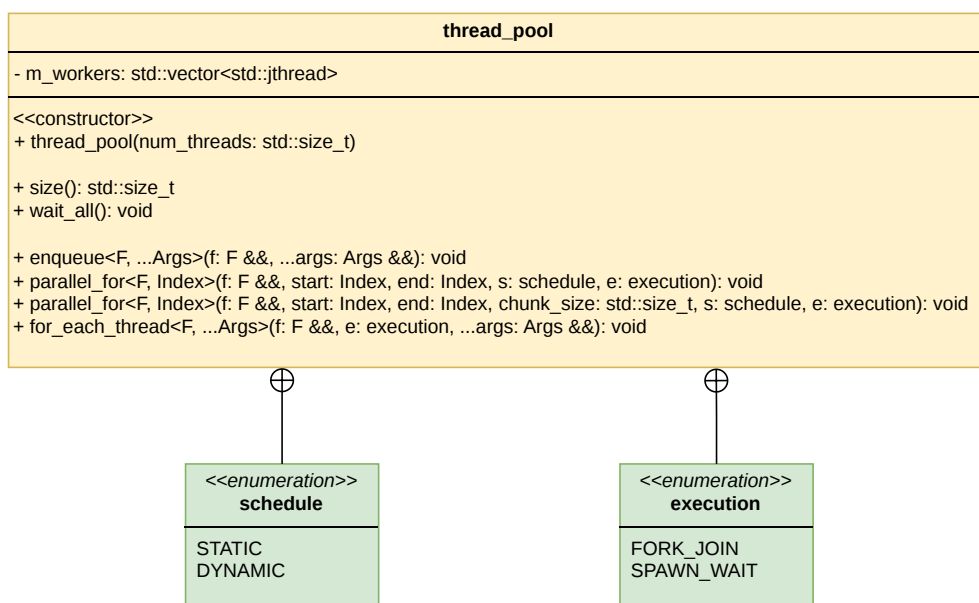
způsoby rozdělení iterací mezi jednotlivá vlákna, a to alespoň statický a dynamický způsob rozdělení. Dále musí rozhraní umožňovat specifikovat velikost bloku přidělovaných iterací a to pro statický i dynamický způsob rozdělení (viz. 3.2.4).

Dále je nezbytné zohlednit i programovací model. OpenMP používá model *fork-join* (viz. 3.2.2), implementace proto musí umožňovat, aby se volající vlákno přidalo k ostatním vláknům v zásobárně pro provádění daného výpočtu. Pokud by vlastní řešení zásobárny úloh nepodporovalo model *fork-join*, pak by se naměřené výsledky mezi sebou špatně porovnávaly, neboť by to znamenalo, že zatímco OpenMP varianta by pracovala s n vlákny, C++ varianta by měla k dispozici $n + 1$ vláken. Zásobárna vláken je však dalším typickým příkladem třídy, kterou by mělo být možné použít i mimo rámec této diplomové práce. Aby byla splněna znovupoužitelnost pro tuto třídu, musí rozhraní umožňovat specifikovat požadavek, jestli má být daná úloha spuštěna v modelu *fork-join*, kdy se volající vlákno bude účastnit výpočtu, nebo v modelu, ve kterém volající vlákno pouze zadá úlohu vláknům v zásobárně a následně bude čekat na jejich dokončení, aniž by se samo účastnilo výpočtu. Tento druhý model bude dále v tomto textu označován jako *spawn-wait*.

Dále je potřeba zajistit podporu pro paralelní oblasti, tj. rozhraní zásobárny vláken musí umožňovat zadat výpočet úlohy pro všechna vlákna, přičemž opět musí být podporován model *fork-join* i *spawn-wait*. Je zřejmé, že pro tyto účely by bylo postačující využít datového paralelismu prostřednictvím smyčky `for` s počtem iterací rovným celkovému počtu vláken, podle toho, zda má být použitý model *fork-join* či *spawn-wait*. Nicméně tento přístup může vést ke špatné srozumitelnosti a v důsledku i udržitelnosti kódu, protože nemusí být zřejmé, proč je pro tento účel použitý právě paralelní cyklus `for`. Proto musí rozhraní poskytovat i možnost zadání úlohy pro všechna vlákna, což bude reprezentovat direktivu pro paralelní oblast z OpenMP.

Na závěr, pro účely znovupoužitelnosti třídy i nad rámec této práce, by rozhraní zásobárny úloh také mělo poskytovat možnost zadat jednorázovou úlohu, kterou si vyzvedne a provede libovolné vlákno v zásobárně. V tomto případě nemusí být nutné, aby volající vlákno čekalo na dokončení výpočtu zadané úlohy a může proto pokračovat dál ve svém výpočtu. Pokud by však volající vlákno potřebovalo čekat na dokončení zadané úlohy, musí mu to rozhraní třídy umožnit.

Na základě výše provedené analýzy byl proveden návrh třídy `thread_pool`, který zachycuje návrhový UML model na obrázku 8.3. Aby bylo možné vkládat úlohy prostřednictvím lhodnotových i rhodnotových argumentů, je úloha určená k paralelizaci metodám předávána ve formě rhodnotové reference na šablonový parametr. Těmito rhodnotovým referencím na šablonový parametr se také říká dopředné reference (z angl. *forwarding references*). Stejným způsobem jsou předávány i argumenty úlohy určené k paralelizaci, avšak aby bylo možné předat libovolný počet argumentů, jsou všechny potřebné metody modelovány jako variadické šablony, kde `template parameter pack` (zkr. `TPP`)



■ **Obrázek 8.3** Návrhový model zásobárny vláken.

zastupuje právě argumenty úlohy, která se má prostřednictvím zásobárny úloh paralelizovat. Způsoby rozdělení iterací v datovém paralelismu a paralelní programovací modely jsou modelovány jako vnořené výčtové typy.

Implementace

Tato kapitola se zabývá popisem implementace aplikací pro sekvenční a paralelní násobení řídkých matic. Nejprve je popsána implementace hešovací tabulky, zásobárny vláken a řídkých matic včetně jejich načítání a zápisu do souboru. Dále je popsán obecný návrh hierarchie tříd, prefixový součet a konverze řídkých matic. Poté je detailně popsáno falešné sdílení a jeho eliminace, a optimalizace kompilátoru umožňující předávání výsledků hodnotou. Na závěr je popsáno CLI rozhraní aplikací a jeho implementace.

Pro účely měření, porovnání a vyhodnocení výsledků byly implementovány tři aplikace. První aplikace provádí násobení řídkých matic sekvenčně, druhá aplikace paralelizuje výpočet za použití knihovny OpenMP a třetí aplikace paralelizuje výpočet prostřednictvím zásobárny vláken bez použití externích knihoven pro paralelizaci.

Všechny tři aplikace byly implementovány ve formě CLI aplikace (z angl. *command-line interface*), tj. aplikace z příkazové řádky. Vzhledem k tomu, že se implementované aplikace od sebe liší pouze paralelizací výpočtu, mají všechny varianty téměř identické rozhraní. Konkrétní podobu tohoto rozhraní včetně jeho implementace popisuje kapitola 9.9.

Implementace dodržuje zásady samovysvětlujícího kódu (z angl. *self-describing code*), který bývá někdy také nazýván samodokumentující kód (z angl. *self-documenting code*) [67]. Zdrojový kód je tedy psán s důrazem na čitelnost a snadnou udržitelnost tím, že metody plní právě jeden úkol a názvy proměnných, metod a tříd jsou voleny tak, aby jejich účel byl zřejmý přímo z názvu. Vyjma komentářů pro popis tříd a jejich metod, které plní účel dokumentace projektu, jsou díky samovysvětlujícímu kódu komentáře přímo v kódu nutné pouze minimálně, např. v místech, kde samotný kód nedokáže záměr dostatečně vyjádřit.

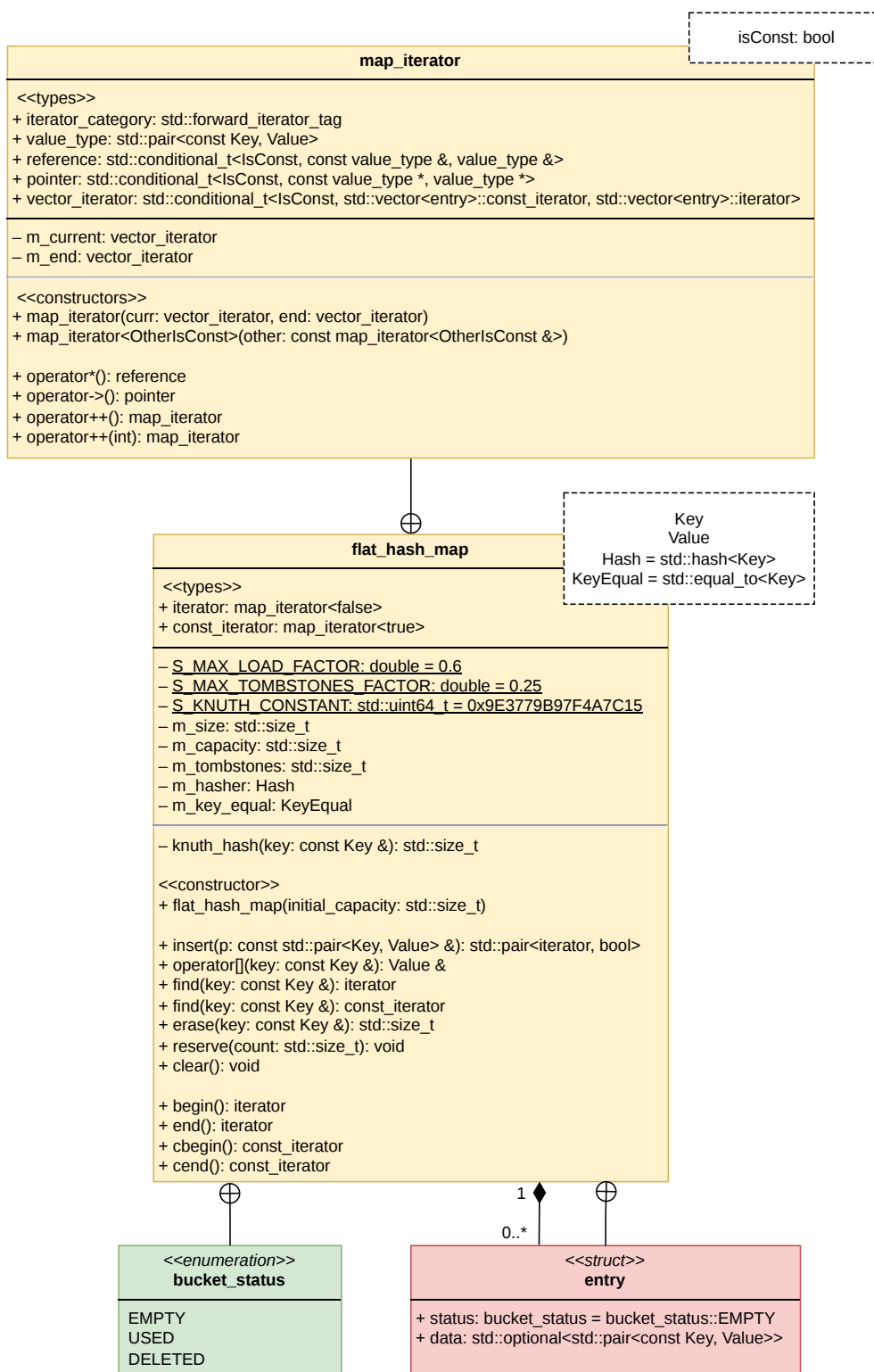
Projekt byl verzován v systému pro správu verzí Git, pro účely automatizace překladač a ověření testů je použit nástroj CMake. Detailní informace o překladač a spuštění aplikací poskytuje uživatelská příručka v příloze A.

9.1 Hešovací tabulka

Hešovací tabulka je typickým příkladem třídy, kterou by mělo být možné využít i nad rámec projektu, pro který byla implementována. Proto pro účely znovupoužitelnosti byla hešovací tabulka `flat_hash_map` s otevřeným adresováním a lineárním přidáváním implementována tak, aby metody této třídy používaly stejné rozhraní jako `std::unordered_map` ze standardní knihovny. Díky stejnému rozhraní implementovaných metod je možné i v jiných projektech hešovací tabulku ze standardní knihovny, která používá hešování s řetězením (viz. 8.2), jednoduše nahradit implementovanou hešovací tabulkou s otevřeným adresováním a to s minimálními změnami ve zdrojovém kódu. Pro snadnější orientaci zachycuje nejdůležitější části implementovaného řešení UML diagram na obrázku 9.1.

Hešovací tabulka byla implementována jako třídní šablona s šablonovými parametry `Key` pro klíč, `Value` pro hodnotu, `Hash` pro hešovací funkci a parametrem `KeyEqual` pro funkci porovnání rovnosti klíčů. Jako výchozí parametr pro hešovací funkci je zvolen `std::hash` ze standardní knihovny. Jak bylo popsáno v 8.2, hešovací funkce vyšší bity součinu s multiplikativní konstantou podle Donalda Knutha očekává na vstupu w -bitové číslo. Aby bylo možné hešovací tabulku použít pro jakékoliv klíče, tj. například i pro uživatelsky definované typy, je zřejmé, že klíč je nezbytné zahešovat dvakrát. Účelem první hešovací funkce je převést klíč na 64-bitové číslo, které je následně vstupem do interní hešovací funkce vyšší bity součinu s multiplikativní konstantou podle Donalda Knutha. Tento postup má dvě zásadní výhody. První výhodou je, že uživatel hešovací tabulky může použít libovolnou hešovací funkci prostřednictvím šablonového parametru, což zvyšuje znovupoužitelnost hešovací tabulky i nad rámec této práce. Druhou výhodou je, že funkční šablona `std::hash` pro celočíselné typy pouze vrací svůj vstup, takže výpočty v rámci této práce nebudou dvojitým hešováním nijak zpomaleny a díky transparentnosti `std::hash` pro celočíselné typy bude interní hešovací funkce pracovat přímo se sloupcovými indexy výstupní matice.

Pro stanovení požadavků na šablonové parametry byly použity koncepty (v angl. *concepts*), které jsou novinkou ve standardu C++20 a představují moderní způsob specifikace a vynucení požadavků na vlastnosti šablonových parametrů. Koncept je množina konkrétních omezení (v angl. *constraints*), která umožňují omezit šablonové parametry na základě specifikované podmínky. Je také vhodné poznamenat, že tato kontrola vlastností šablonových parametrů probíhá v čase překladu, jak plyne ze způsobu fungování C++ šablon, a má proto nulovou režii při běhu programu. Pro hešovací tabulku `flat_hash_map` byly specifikovány požadavky, aby výstup uživatelem dodané hešovací funkce byl převoditelný na datový typ `std::size_t` a výstup šablonového parametru určující funkci pro porovnání dvou klíčů byl převoditelný na datový typ `bool`. Tento koncept byl pojmenován `hashable_key` a jeho podobu ukazuje výpis kódu 9.1.



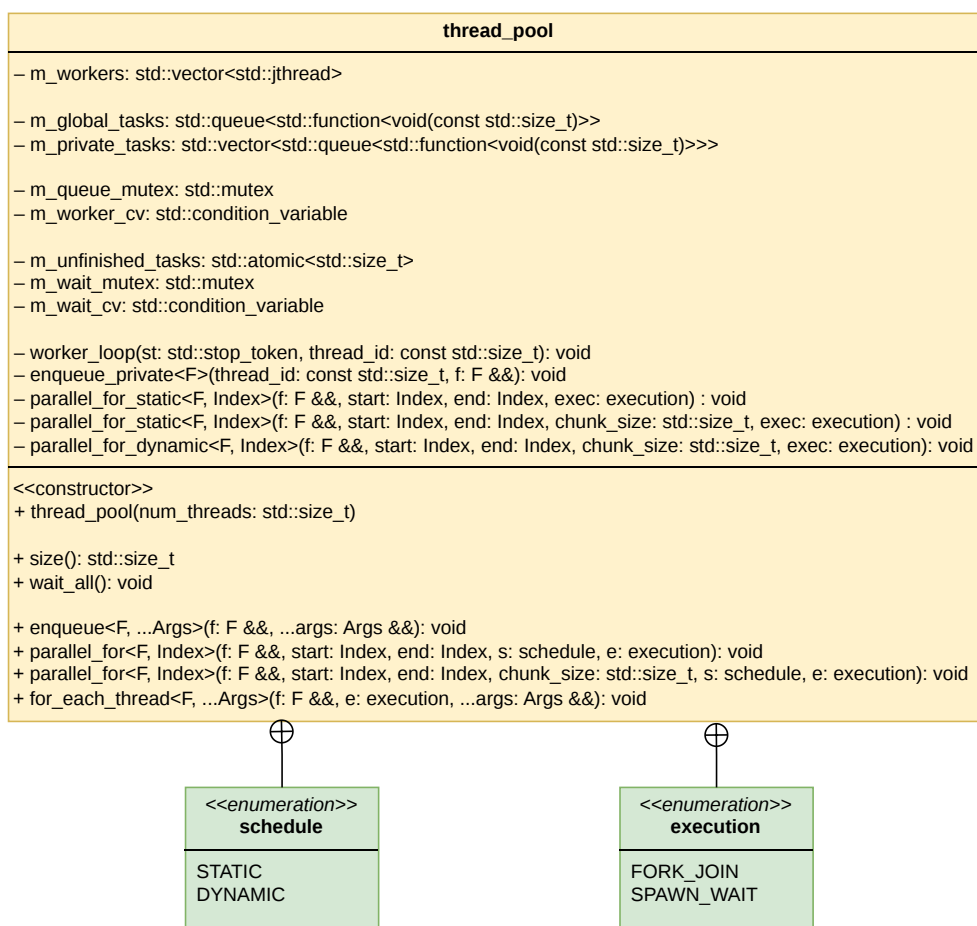
■ **Obrázek 9.1** UML diagram klíčových částí hešovací tabulky.

```
template<typename Key, typename Hash, typename KeyEqual>
concept hashable_key = requires(Key key,
                                Hash hash,
                                KeyEqual key_equal) {
    { hash(key) } -> std::convertible_to<std::size_t>;
    { key_equal(key, key) } -> std::convertible_to<bool>;
};
```

■ **Výpis kódu 9.1** C++ koncept `hashable_key` pro šablonové parametry hešovací tabulky s otevřeným adresováním.

Příhrádky hešovací tabulky jsou reprezentovány strukturou `entry`, která uchovává stav aktuální příhrádky a dále pár **Key-Value**, který je zapouzdřen uvnitř objektu `std::optional`. Tato třídní šablona je novinkou ze standardu C++17 a spravuje volitelně obsaženou hodnotu, tj. hodnotu, která může, ale nemusí být přítomna. Zapouzdření klíče a hodnoty v objektu `std::optional` je nezbytné z důvodu znovupoužitelnosti hešovací tabulky, protože ačkoliv v rámci této práce budou hodnotami celá nebo reálná čísla, obecně nelze o šablonovém parametru `Value` nic předpokládat. Je tedy možné, že konstrukce hodnoty může být výpočetně velmi náročná, proto je vhodné konstrukci objektu oddálit až na nejzazší možnou dobu, kterou je vložení nového klíče do tabulky. Navíc pro účely implementace metody `erase()` pro vymazání páru klíče a hodnoty z tabulky je potřeba zajistit korektní destrukci klíče i hodnoty, aby tyto objekty mohly například uvolnit své alokované zdroje. Třídní šablona `std::optional` pro tyto účely disponuje metodou `reset()`, která zajistí bezpečné volání destruktoru klíče i hodnoty. Klíč `Key` je v příhrádce uložen jako konstantní, aby nebylo možné prostřednictvím iterátoru (viz. dále) modifikovat obsah klíče, čímž by se hešovací tabulka mohla dostat do nekorektního stavu. Příhrádky hešovací tabulky jsou spravovány uvnitř kontejneru `std::vector`.

Pro účely znovupoužitelnosti byl také implementován dopředný iterátor (z angl. *forward iterator*) ve formě vnořené třídní šablony. Záměrem bylo vytvořit oddělené třídy pro konstantní i nekonstantní iterátor, avšak oba tyto iterátory by měly stejnou logiku a měly by proto i stejný kód, což vede ke špatné udržitelnosti. Proto byl iterátor implementován jako třídní šablona, jejíž parametr určuje, zda se jedná o konstantní nebo nekonstantní iterátor. Zároveň byl implementován i konverzní konstruktor, který umožňuje inicializovat konstantní iterátor nekonstantním iterátorem. Pro konverzní konstruktor bylo zároveň implementováno omezení, které zaručuje, že nebude možné inicializovat nekonstantní iterátor konstantním iterátorem. Návrátové typy metod iterátoru, které jsou odlišné pro konstantní a nekonstantní iterátor, jsou nastaveny prostřednictvím šablonové metafunkce `std::conditional_t`.



■ Obrázek 9.2 UML diagram zásobárny vláken.

9.2 Zásobárna vláken

Pro implementaci zásobárny vláken `thread_pool` je potřeba zajistit správu vláken v zásobárně, frontu úloh ke zpracování a synchronizaci vláken pro ko-rektní vkládání a vyzvedávání úloh z fronty. Pro lepší názornost byl návrhový model z kapitoly 8.3 rozšířen na diagram tříd, který zachycuje všechny třídní proměnné a metody. Tento diagram je k vidění na obrázku 9.2.

Pro správu vláken byl zvolen kontejner `std::vector` a pro reprezentaci vlákna byla zvolena třída `std::jthread` ze standardu C++20. Jak bylo popsáno v kapitole 3.1.2, v destruktoru této třídy je automaticky volána metoda `join`, čímž odpadá nutnost volat tuto metodu explicitně. Nicméně ačkoliv byla pro reprezentaci vláken zvolena právě tato třída, je z důvodu udržitelnosti v destruktoru zásobárny vláken pro každé vlákno metoda `join` explicitně volána. Při destrukci objektu `thread_pool` by totiž mohla nastat situace, kdy třídní proměnné, které zajišťují synchronizaci vláken a spravují úlohy ve frontě, bu-

dou destruovány dříve, než samotná výpočetní vlákna. Taková situace vede k nedefinovanému chování a mohlo by nastat uváznutí programu (viz. 3.1.3). Vzhledem k tomu, že je standardem C++ zaručeno, že třídní proměnné jsou destruovány v opačném pořadí jejich deklarace, bylo by možné deklarovat `std::vector<std::jthread>` jako poslední třídní proměnnou, čímž by bylo zaručeno, že vlákna budou destruována jako první a nebylo by tak nutné explicitně volat metodu `join` v destruktoru zásobárny vláken. Je však zřejmé, že taková implementace vede ke špatné udržitelnosti, neboť pořadí deklarace třídních proměnných by nemělo ovlivňovat korektnost programu. Z těchto důvodů je proto pro každé vlákno metoda `join` zavolána explicitně, čímž je zaručeno, že nedojde k uváznutí nebo nedefinovanému chování programu při destrukci zásobárny vláken v důsledku změny pořadí deklarace třídních proměnných.

První verze implementace zásobárny vláken používala pouze jednu frontu úloh, do které byly úlohy ukládány ve formě objektu `std::function<void()>`. Toto řešení se však ukázalo jako nedostatečné, neboť umožňuje pouze dynamické rozdělení paralelního `for` cyklu a neumožňuje použít statické rozdělení iterací jednotlivým vláknům. Bylo proto nezbytné přidat pro každé vlákno jeho vlastní privátní frontu úloh pro statické rozdělení iterací `for` cyklu, zatímco globální fronta pro dynamické rozdělení a pro vložení obecné úlohy zůstala společná pro všechna vlákna.

Dále se ukázala jako velmi problematická samotná reprezentace úloh formou objektů `std::function<void()>`. Tato reprezentace na rozdíl od konstruktů v OpenMP totiž vůbec neumožňuje uvnitř úlohy pracovat s číselným identifikátorem vlákna, které danou úlohu zpracovává. Bylo proto nezbytné implementovat způsob, který úlohám umožní pracovat s číselnou reprezentací vlákna, které ji vykonává.

Prvním řešením bylo každé úloze, která je vložena do fronty vláken, pevně přiřadit číselný identifikátor, který bude simulovat identifikátor vlákna, jež tuto úlohu provádí. Toto řešení se však záhy ukázalo jako nepoužitelné, neboť by mohla nastat situace, kdy si dvě vlákna ze společné globální fronty úloh vyzvednou úlohy se stejným identifikátorem a budou tyto úlohy provádět souběžně. Vzhledem k tomu, že číselný identifikátor vlákna se často používá jako index pro přístup k datovým strukturám, mohlo by toto řešení vést k časově závislým chybám (viz. 3.1.1). Z těchto důvodů byla změněna reprezentace úloh na `std::function<void(const std::size_t)>`, která jako argument přijímá číslo vlákna, které bude danou úlohu vykonávat. Číselný identifikátor vlákna pak dodává přímo samotné vlákno, které tuto úlohu vykonává.

Každému vláknu je při jeho vytvoření v konstruktoru zásobárny úloh předána funkce `worker_loop`, která reprezentuje smyčku, ve které vlákno čeká na vložení nové úlohy do globální nebo privátní fronty úloh. Tato funkce jako první parametr přijímá `std::stop_token`, prostřednictvím kterého jsou v destruktoru zásobárny úloh jednotlivá vlákna informována o ukončení výpočtu. Druhým parametrem pak je právě číselné označení vlákna, které každé vlákno dostane pevně přidělené při jeho vytvoření. Pokud je v zásobárně n vláken,

```

template<typename F, typename ...Args>
requires std::invocable<F, std::size_t, Args...>
void thread_pool::enqueue(F && f, Args && ...args) {
    {
        std::lock_guard<std::mutex> lock(m_queue_mutex);
        m_global_tasks.push(
            [task = std::bind(std::forward<F>(f),
                              std::placeholders::_1,
                              std::forward<Args>(args)...)]
            (std::size_t thread_id) {
                task(thread_id);
            });
    }
    ++m_unfinished_tasks;
    m_worker_cv.notify_one();
}

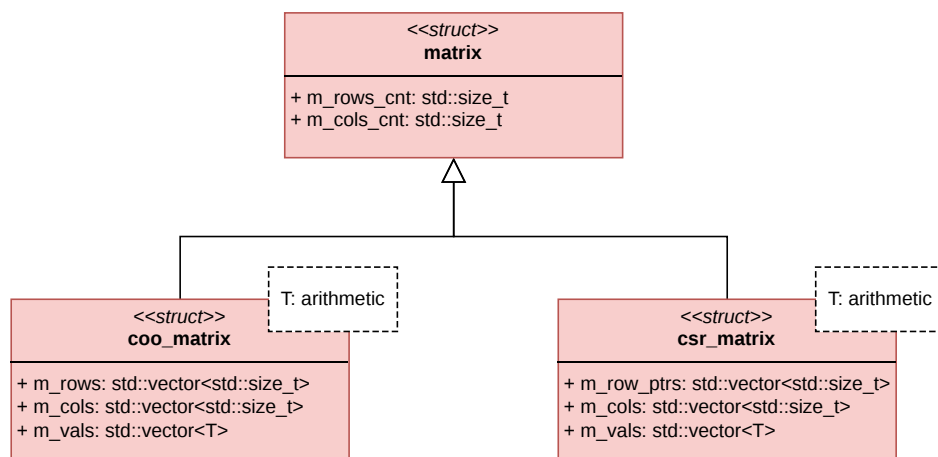
```

■ **Výpis kódu 9.2** Implementace metody `enqueue` pro vložení obecné úlohy do zásobárny vláken.

pak jsou vláknům přiřazeny identifikátory $1 \dots n$, neboť identifikátor 0 je rezervován pro volající vlákno za účelem simulace master vlákna v OpenMP.

Dále bylo potřeba vyřešit, jakým způsobem bude do výpočtu zapojeno volající vlákno v případě modelu *fork-join*. Jednak je zřejmé, že pro tento účel je nezbytné, aby zásobárna vláken alokovala privátní frontu úloh i pro volající vlákno, neboť jinak by se nemohlo účastnit výpočtu statického rozdělení iterací `for` cyklu. První řešení pro zapojení volajícího vlákna do výpočtu využívalo samostatnou funkci `join_workers`, která zajišťovala odebrání úloh z front zásobárny. Tato funkce však sdílela značnou část kódu s funkcí `worker_loop` a lišila se především tím, že neobsahovala `std::stop_token` hlavního vlákna. Tento způsob je očividně špatně udržitelný, neboť je při změnách implementace třídy `thread_pool` potřeba modifikovat kód na dvou různých místech. Z toho důvodu byla i pro hlavní vlákno použita funkce `worker_loop`, přičemž `std::stop_token` je pro volající vlákno vytvořen explicitně a je rovnou nastaven na požadavek o zastavení výpočtu vlákna. Tím je zajištěno, že jakmile hlavní vlákno dokončí svou účast při vykonávání úloh v zásobárně, tak se rovnou vrátí z této funkce.

Úlohy jsou do front vkládány prostřednictvím anonymní lambda funkce, která přijímá výše popsany číselný identifikátor vlákna, které bude úlohu vykonávat, a tento identifikátor je následně v těle lambda funkce předán uživatelské funkci společně s jejími argumenty. Ukázku tohoto postupu je možné vidět ve výpisu 9.2, který ukazuje implementaci vložení obecné úlohy do globální fronty vláken. Z výpisu kódu je zřejmé, že uživatelská funkce je se svými argumenty svázána pomocí funkce `std::bind`, přičemž samotná uživatelská funkce i její argumenty jsou za účelem zachování hodnotové kate-



■ **Obrázek 9.3** UML diagram struktur pro COO a CSR formáty řídkých matic.

gorie předány technikou *perfect forwarding* prostřednictvím funkční šablony `std::forward`. K zajištění, aby takto svázaná funkce byla později volatelná s identifikátorem vlákna, které bude tuto úlohu provádět, je použitý zástupný objekt `std::placeholders::_1`. Ověření, zda je uživatelem poskytnutá funkce volatelná s identifikátorem vlákna a poskytnutými argumenty, je zajištěno pomocí omezení prostřednictvím klauzule `requires`.

9.3 Řídké matice a jejich načítání a zápis

Jedním z cílů této práce je implementovat násobení řídkých matic pro alespoň dva různé formáty. Jako tyto formáty byly zvoleny CSR a COO (viz. 6), přičemž oba tyto formáty byly implementovány tak, aby co nejvíce odpovídaly jejich definici. Pro lepší názornost je k dispozici UML diagram obou formátů na obrázku 9.3.

Protože oba formáty pouze uchovávají data a neposkytují žádnou další funkcionalitu, jsou reprezentovány prostřednictvím struktury. Pro oba formáty byla implementována společná rodičovská třída `matrix`, která reprezentuje obecnou matici a uchovává pouze počet řádků a sloupců této matice. Implementace COO formátu používá tři proměnné typu `std::vector`, jež uchovávají řádkové indexy, sloupcové indexy a hodnoty nenulových prvků. CSR formát je implementován analogicky, pouze místo řádkových indexů uchovává řádkové ukazatele.

Nabízí se otázka, jakým datovým typem reprezentovat nenulové prvky matice. Počáteční implementace nenulové prvky reprezentovala pouze datovým typem `double`. Takové řešení je však nedostatečné, neboť neumožňuje využívat všechny datové typy podporované souborovým formátem Matrix Market a navíc uživateli neumožňuje vybrat datový typ pro nenulové prvky explicitně. Pevné navázání na konkrétní typ by proto významně omezilo možnosti zno-

```
template<typename T>
constexpr bool is_complex_v = false;

template<std::floating_point T>
constexpr bool is_complex_v<std::complex<T>> = true;

template<typename T>
concept complex_number = is_complex_v<T>;

template<typename T>
concept arithmetic = std::integral<T> ||
                    std::floating_point<T> ||
                    complex_number<T>;
```

■ **Výpis kódu 9.3** Implementace konceptů pro komplexní a aritmetické datové typy.

vupoužitelnosti těchto datových struktur, zejména v situacích, kdy je žádoucí pracovat s odlišnou numerickou přesností či celočíselnými typy. Z těchto důvodů byl pro datový typ nenulových prvků zvolen šablonový parametr typu `T`, čímž je zajištěna obecnost a znovupoužitelnost implementace. Datové struktury obou formátů se tak změnilý na šablony struktur.

Je však zřejmé, že je potřeba pomocí omezení určit, které datové typy mohou být substituovány za šablonový parametr. Aby implementace umožňovala pracovat se všemi datovými typy podporovanými formátem Matrix Market, musí být možné za šablonový parametr substituovat celočíselný, reálný i komplexní datový typ. Z těchto důvodů bylo nutné implementovat koncept `arithmetic`, který ověří, zda daný datový typ je celočíselný, reálný nebo komplexní. Zatímco pro celočíselné a reálné datové typy jsou od verze C++20 k dispozici koncepty `std::integral` a `std::floating_point`, pro komplexní datový typ takový koncept k dispozici není a je proto nezbytné jej implementovat ručně.

Pro implementaci konceptu `complex_number` je zapotřebí implementovat šablonovou metafunkci, která na základě svého šablonového parametru určí, zda je daný datový typ komplexní či nikoliv. Datový typ `std::complex` ze standardní knihovny však povoluje jako reprezentaci reálné a imaginární části komplexního čísla pouze datové typy s plovoucí řádovou čárkou. [62, 63] Je proto zřejmé, že je nutné této šablonové metafunkci omezit její vlastní šablonový parametr prostřednictvím konceptu `std::floating_point`. Tuto šablonovou metafunkci, koncept `complex_number` a koncept `arithmetic` ukazuje výpis kódu 9.3.

Jako souborový formát byl pro účely této práce zvolen formát Matrix Market (viz. 6.2) a bylo proto nezbytné implementovat načtení matice ve formátu Matrix Market a její uložení do struktury popsané výše, resp. zápis matice v COO nebo CSR formátu do souborového formátu Matrix Market. Nabízí se

možnost implementovat tuto funkcionalitu ručně, avšak formát Matrix Market je velmi obsáhlý, jak je popsáno v kapitole 6.2, a vlastní implementace by tak podporovala pouze určitou podmnožinu celého Matrix Market formátu. Z těchto důvodů bylo pro načítání i zápis matic použito existující řešení v podobě knihovny `fast_matrix_market` [68]. Tato knihovna využívá paralelismus pro čtení i zápis matic a navíc podporuje kompletní formát Matrix Market, tj. veškeré datové typy včetně komplexních čísel, všechny symetrické struktury matic a oba způsoby reprezentace nenulových prvků. Použitím této knihovny je tedy zaručeno, že bude načtena jakákoliv matice v korektním Matrix Market formátu, tedy například jakákoliv matice z kolekce SuiteSparse Matrix Collection [24].

Jak bylo popsáno výše, řešení, které reprezentuje nenulové prvky každé matice stejným datovým typem, je nedostatečné. Je proto zřejmé, že i řešení, které každou matici uloženou ve formátu Matrix Market načte do struktury s předem pevně stanoveným datovým typem (např. `coo_matrix<double>`), je nevyhovující. Kvalitní řešení by mělo vybrat datový typ na základě informace o datovém typu nenulových prvků, která je uvedena v hlavičce souboru Matrix Market. Zvolená knihovna `fast_matrix_market` [68] pro tyto účely poskytuje možnost načíst pouze hlavičku Matrix Market souboru. Řešení implementované v rámci této práce proto nejprve načte hlavičku Matrix Market souboru a teprve na základě údaje o datovém typu nenulových prvků použije odpovídající instanci šablony struktury `coo_matrix`. Pokud je tedy v Matrix Market souboru uvedený datový typ `real`, je matice načtena do struktury `coo_matrix<double>`. Pokud je v Matrix Market souboru uvedený datový typ `complex`, je matice načtena do struktury `coo_matrix<std::complex>`, atd.

Knihovna pro načítání matic `fast_matrix_market` [68] navíc umožňuje automatický `std::complex` up-cast, tedy je například možné matici s celočíselnými prvky načíst do struktury datového typu `std::complex<double>`. Tato funkcionalita byla využita pro implementaci násobení matic dvou různých datových typů. Nejprve je pro každou ze vstupních matic **A** a **B** načtena jejich Matrix Market hlavička, poté je určen jejich společný datový typ a na závěr jsou obě vstupní matice načteny do struktur tohoto společného datového typu, který zároveň určuje i datový typ výstupní matice **C**. Pokud jsou například na vstupu Matrix Market soubory s maticemi s datovými typy `real` a `complex`, jsou obě matice načteny do struktur `coo_matrix<std::complex>`, která je současně i reprezentací výstupní matice **C**. Řešení implementované v rámci této práce dále tento datový typ výstupní matice využívá k určení datového typu, který bude specifikován ve výstupním Matrix Market souboru. Pokud je například výstupní matice **C** datového typu `double`, bude do výstupního Matrix Market souboru zapsán datový typ `real`, pokud je výstupní matice datového typu `std::int64_t`, bude do výstupního Matrix Market souboru zapsán datový typ `integer`, atd.

9.4 Hierarchie tříd

V rámci této diplomové práce je mnoho tříd implementováno ve třech variantách. První je sekvenční varianta, druhá je paralelní varianta za použití OpenMP a třetí varianta paralelizuje výpočet s využitím implementované zásobárny vláken (viz. 9.2). Z tohoto důvodu byla v první implementaci použita třídní hierarchie, která pro každé tři výše popsané varianty dané třídy používala společnou rodičovskou čistě abstraktní třídu, která bývá v jiných programovacích jazycích často také označována jako rozhraní (z angl. *interface*). V rodičovské čistě abstraktní třídě byl tedy definován kontrakt prostřednictvím virtuálních metod a každá varianta následně tyto metody implementovala ve své vlastní třídě.

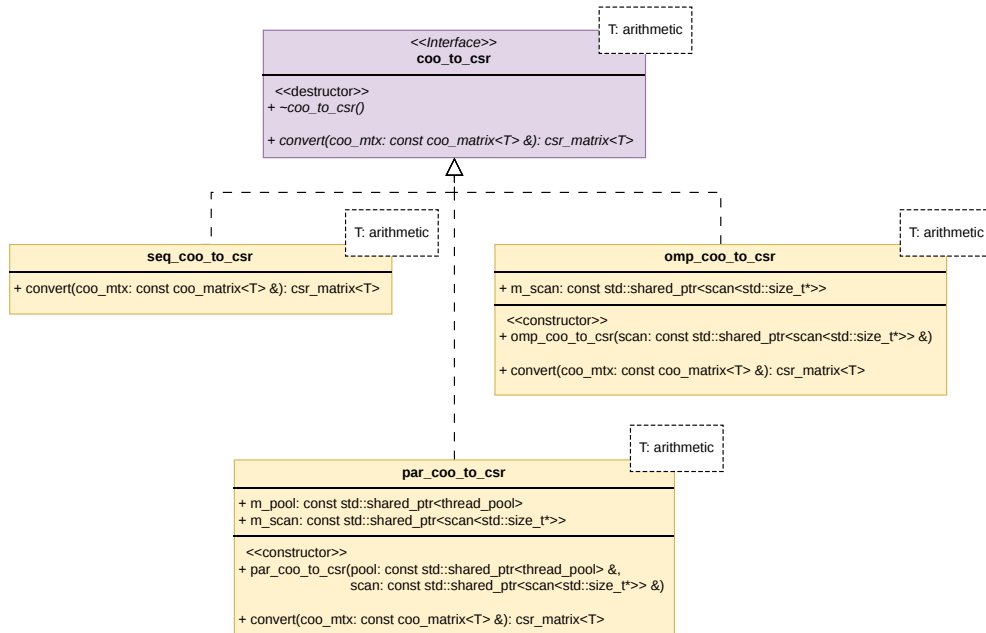
Toto řešení se zdálo být optimální pouze ve stavu, kdy datové struktury COO a CSR matic nepoužívaly šablonový parametr. Po přepracování reprezentací těchto maticových formátů na šablony struktur (viz. 9.3) se toto řešení ukázalo jako velmi problematické. Každou virtuální metodu, která přijímala COO nebo CSR matici jako parametr, bylo totiž nutné přepracovat na generickou virtuální metodu. Jazyk C++ však nepodporuje virtuální metody se šablonovým parametrem, což je pro výše popsanou třídní hierarchii naprosto zásadní komplikace.

Prvním řešením tohoto problému bylo převést všechny třídy v popsané třídní hierarchii na třídní šablony. Tím, že byl generický parametr přesunut z třídních metod do samotné třídy, bylo možné virtuální metody bez omezení používat, neboť již nebyly generické. Ukázkou tohoto řešení na příkladu tříd pro konverzi COO matic do formátu CSR ukazuje UML diagram tříd na obrázku 9.4.

Záhy se však ukázalo, že tento přístup má tři zásadní problémy. Prvním problémem je sémantický význam takového řešení. Jedním z účelů použití třídních šablon je, že se daná třída bude chovat pro různé šablonové parametry odlišným způsobem. Toto však v případě výše popsané třídní hierarchie není pravda, neboť dané třídy se budou chovat identicky bez ohledu na to, zda jsou instanciovány pro datový typ `double` nebo `std::uint32_t`. Takové řešení proto očividně vede ke špatné udržitelnosti, neboť zhoršuje čitelnost kódu a zkresluje jeho sémantický význam.

Druhý problém je, že je nezbytné pro každý datový typ vytvořit separátní instanci této třídy. Například pro převod COO matice typu `double` a COO matice typu `std::uint32_t` do formátu CSR to znamená vytvořit dvě separátní instance třídy `coo_to_csr`, ačkoliv je s oběma maticemi v rámci konverze zacházeno stejným způsobem bez ohledu na jejich datový typ.

Třetím problémem je, že je nutné datový typ specifikovat explicitně, neboť je potřeba každou třídu před voláním jejích metod nejprve instanciovat, což znamená, že nelze použít automatickou dedukci šablonových argumentů pro metody těchto tříd. Tento postup je velmi nepraktický, jak je například vidět na UML diagramu na obrázku 9.4, kde je ve třídě `omp_coo_to_csr` jako členská

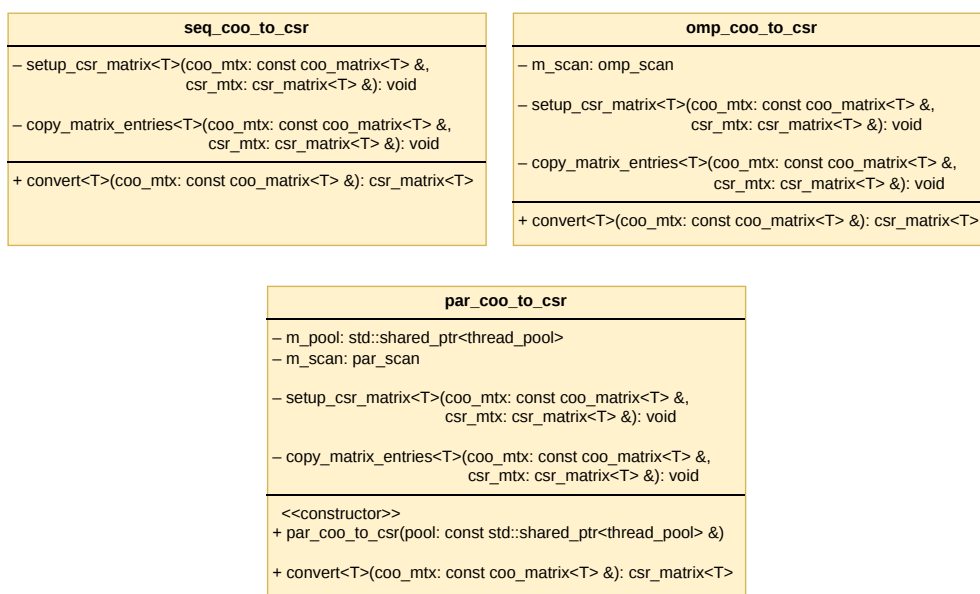


■ **Obrázek 9.4** UML diagram původní třídní hierarchie na příkladu tříd pro konverzi matic.

proměnná chytrý ukazatel na instanci typu `scan<std::size_t*>`, která zajišťuje prefixový součet (viz. 9.5). Přitom třída pro výpočet prefixového součtu je typickým zástupcem třídy, kterou by mělo být možné použít pro kontejnery jakéhokoli aritmetického typu právě prostřednictvím automatické dedukce šablonových argumentů v metodách této třídy.

Jedinou výhodou této třídní hierarchie je, že lze kód, který pracuje pouze s rozhraním daných tříd, dodat jakoukoliv instanci ze tří variant dané třídy. Nicméně při zohlednění faktu, že se OpenMP direktivy nemají používat společně s jinými knihovnami pro vytváření a správu vláken, přestává i tato vlastnost být výhodou. Z těchto důvodů byla popsaná třídní hierarchie opuštěna a projekt byl přepracován tak, aby třídy mohly používat šablonové metody, tj. není použita společná rodičovská čistě abstraktní třída. Tento přístup umožňuje danou třídu použít pro matice s jakýmkoliv datovým typem reprezentujícím nenulové prvky této matice, a díky automatické dedukci šablonových argumentů ani není nutné šablonový parametr uvádět explicitně. Přepracované třídy pro konverzi COO matice do CSR formátu ukazuje UML diagram na obrázku 9.5.

Stále jsou však dodržovány principy kvalitního softwarového návrhu, například pokud všechny tři varianty dané třídy používají metodu, jejíž kód je ve všech variantách totožný, pak byla pro tyto třídy vytvořena společná rodičovská třída, do které byla metoda přesunuta.



■ **Obrázek 9.5** UML diagram tříd pro konverzi matic po přepracování na šablonové metody.

9.5 Prefixový součet

V implementaci všech tří variant násobení řídkých matic je často používanou operací prefixový součet. Pro sekvenční verzi nebylo nutné implementovat vlastní prefixový součet, neboť je dostupný ve standardní knihovně jazyka C++ v podobě funkční šablony `std::inclusive_scan`. Jak bylo popsáno v kapitole 3.1.6, od C++17 je tento prefixový součet dostupný i v paralelní verzi prostřednictvím specifikace způsobu provádění `std::execution::par`. Nicméně pro vědecké výpočty není tato paralelní verze použitelná, neboť, jak bylo popsáno v 8.3, není zaručeno, že výpočet tohoto algoritmu skutečně bude proveden paralelně a navíc není možné specifikovat počet vláken pro výpočet ani zpětně zjistit, kolik vláken bylo pro výpočet použito. Je tedy zřejmé, že pro paralelní varianty SpGEMM výpočtu musí být pro účely této práce použito jiné řešení.

V případě OpenMP varianty je možné použít prefixový součet, který nabízí přímo OpenMP API od verze 5.0. Tato verze přidává podporu pro inkluzivní i exkluzivní paralelní prefixový součet prostřednictvím klauzule `reduction` s modifikátorem `inscan`. Uvnitř cyklu je pak potřeba prostřednictvím direktivy `scan` specifikovat, zda se jedná o inkluzivní nebo exkluzivní prefixový součet. Paralelní prefixový součet v OpenMP byl pro účely této práce implementován v samostatné třídě jako šablonová metoda `inclusive`, která jako své parametry přijímá iterátory, podobně jako je tomu u `std::inclusive_scan` ze standardní knihovny, aby bylo možné paralelní prefixový součet vypočítat pro data uložená v jakémkoliv kontejneru. Toto řešení ukazuje výpis kódu 9.4. Aby bylo možné přistupovat k prvkům prostřednictvím operátoru `indexo-`

vání (tj. operátoru náhodného přístupu), je na šablonový parametr reprezentující iterátor aplikováno omezení konceptem `std::random_access_iterator`, který je dostupný od verze C++20.

Úplná specifikace OpenMP verze 5.0 sice ještě není implementována ani v nejnovějších verzích překladačů GCC a Clang, které jako podporovanou verzi uvádějí OpenMP 4.5. Nicméně jak je uvedeno v [69], OpenMP 5.0 je částečně podporováno kompilátorem GCC již od verze 9, přičemž GCC 15 implementuje téměř veškerou funkcionalitu OpenMP 5.0. Obdobně překladač Clang verze 21 podporuje téměř veškerou specifikaci OpenMP 5.0. Pro automatickou kompilaci a ověření testů (viz. kapitola 10) byl použit Docker obraz kompilátoru GCC ve verzi 13. Tato verze kompilátoru je zároveň k dispozici i na fakultním výpočetním klastru ClusterFIT, na kterém bylo provedeno měření (viz. kapitola 11). V rámci této práce bylo otestováno, že paralelní prefixový součet z OpenMP 5.0 je podporován překladačem GCC již ve verzi 11 a je tedy zaručeno, že paralelní prefixový součet z OpenMP 5.0 lze ve výše zmíněných prostředích použít.

Z důvodů popsaných výše je pro variantu paralelního SpGEMM násobení implementovaném v C++ nezbytné implementovat vlastní paralelní prefixový součet. Pro tento účel byla použita třífázová varianta paralelního prefixového součtu, která je popsána ve knize [70]. Pole vstupních dat je nejprve rozděleno na p bloků, kde p je počet vláken. V první fázi každé vlákno vypočítá prefixový součet nad svým blokem dat a uloží poslední hodnotu, která reprezentuje součet hodnot v bloku, do pole částečných součtů. Ve druhé fázi každé vlákno vypočítá celkový offset pro svůj konkrétní blok sečtením odpovídajících hodnot v poli částečných součtů. Ve třetí fázi pak každé vlákno přičte svůj offset ke každému prvku ve svém bloku. Postup tohoto algoritmu ilustruje obrázek 9.6.

Tato C++ varianta paralelního prefixového součtu byla stejně jako OpenMP varianta implementována prostřednictvím šablonové metody, která jako parametry přijímá iterátory splňující koncept `std::random_access_iterator`. Je zřejmé, že před výpočtem druhé fáze je potřeba vlákna synchronizovat, neboť pro výpočet offsetu je nezbytné vyčkat, než každé vlákno zapíše svoji hodnotu do pole částečných součtů. Nabízí se pro synchronizaci použít bariéru `std::barrier`, nicméně jak bylo popsáno v kapitole 3.1.7, bariéra rozšiřuje funkcionalitu závory, přičemž tato rozšíření by pro paralelní prefixový součet zůstala nevyužita. Z toho důvodu byla pro synchronizaci vláken v tomto případě použita závora `std::latch`, která je ve standardní knihovně dostupná od C++20.

9.6 Konverze řídkých matic

Algoritmus pro násobení řídkých matic, který byl popsán v kapitole 8.1, očekává na svém vstupu řídké matice ve formátu komprimovaných řádků CSR. Jedním z cílů této práce však je implementovat SpGEMM násobení pro alespoň dva různé formáty. Jako druhý formát byl zvolen souřadnicový formát

```

template<std::random_access_iterator InputIt,
        std::random_access_iterator OutputIt>
void omp_scan::inclusive(InputIt first,
                        InputIt last,
                        OutputIt dest) const {

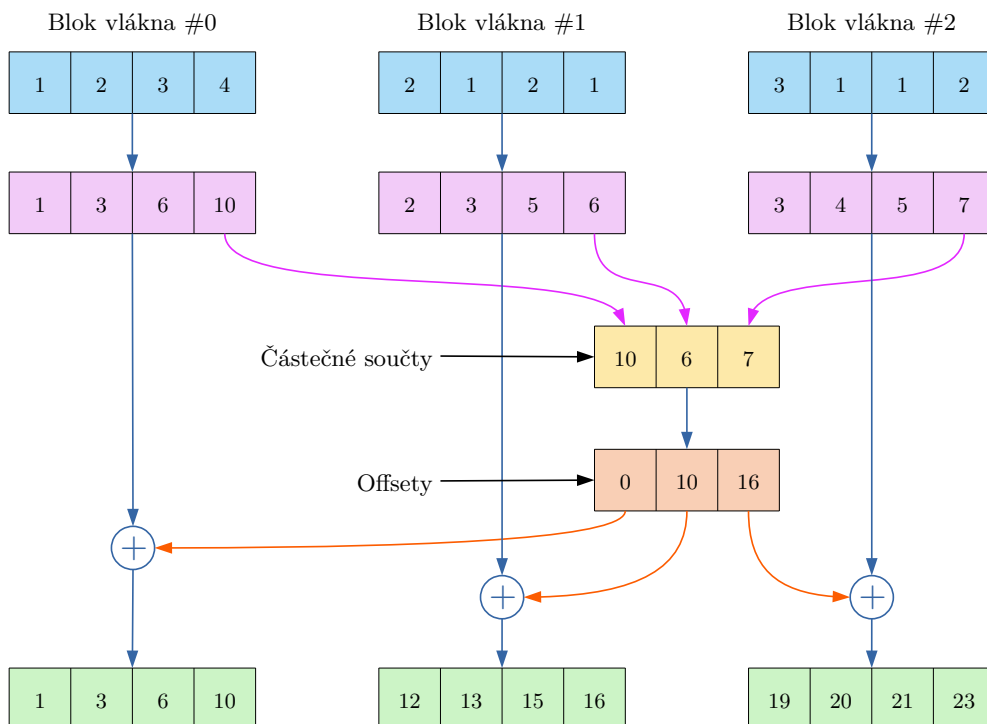
    using T = std::iter_value_t<InputIt>;
    const std::size_t input_size = std::distance(first, last);
    T sum{};

    #pragma omp parallel for reduction(incscan, +:sum)
    for (std::size_t i = 0; i < input_size; ++i) {
        sum += first[i];

        #pragma omp scan inclusive(sum)
        dest[i] = sum;
    }
}

```

■ **Výpis kódu 9.4** Implementace paralelního prefixového součtu s využitím konstrukce z OpenMP 5.0.



■ **Obrázek 9.6** Ukázka třífázového paralelního prefixového součtu. Inspirováno ilustrací z [70].

COO a je proto nezbytné implementovat třídu, která bude převádět COO matici do formátu CSR. Navíc, protože souborový formát Matrix Market ukládá matice pouze v hustém nebo koordinačním formátu (viz. kapitola 6.2.3), tak i knihovna `fast_matrix_market`, která byla použita pro načítání matic ze souboru (viz. kapitola 9.3), podporuje načítání matic pouze do hustého nebo COO formátu. Pro účely SpGEMM algoritmu popsaného v kapitole 8.1 je proto implementace konvertoru COO matice do CSR formátu nezbytná.

První verze implementace používala k převodu COO matice do CSR formátu řazení. Konkrétně využívala permutační vektor velikosti počtu nenulových prvků, který byl seřazen podle řádkových a sloupcových indexů nenulových prvků v COO matici. Následně byly překopírovány nenulové prvky a sloupcové indexy do struktury CSR matice na pozici určenou seřazeným permutačním vektorem. Na závěr bylo vypočítáno pole četností řádkových indexů, které po výpočtu prefixového součtu odpovídalo poli ukazatelů v CSR matici.

Je zřejmé, že složitost takového algoritmu je $O(nnz(\mathbf{A}) \cdot \log nnz(\mathbf{A}))$, kde $\mathbf{A}$ označuje vstupní matici v COO formátu. Výstupní matice v CSR formátu tak bude mít své nenulové prvky v každém řádku seřazené podle sloupcového indexu. Jak však bylo popsáno v kapitole 8, významnou výhodou zvoleného algoritmu pro násobení řídkých matic je, že nevyžaduje, aby vstupní matice měly v rámci svých řádků nenulové prvky seřazené podle sloupcových indexů. Po konverzi COO matice do CSR formátu algoritmem popsaným výše by tato výhoda byla ztracena a zůstala by nevyužita.

Proto byl algoritmus pro konverzi přepracován tak, aby řazení nebylo během konverze vůbec použito. Toto vylepšené řešení nejprve vypočítá pole četností řádkových indexů a nad tímto polem provede prefixový součet pro výpočet pole řádkových ukazatelů formátu CSR. Následně jsou sloupcové indexy a nenulové prvky COO matice překopírovány tak, že správná pozice těchto dat v polích CSR matice je určena na základě informací získaných z vypočteného pole řádkových ukazatelů CSR matice. Pole řádkových ukazatelů totiž obsahuje informaci, na jakém indexu začínají data daného řádku a tuto informaci lze proto využít k zápisu sloupcového indexu a odpovídajícího nenulového prvku na správnou pozici, přičemž po jejich zápisu musí být daný řádkový ukazatel inkrementován, aby ukazoval na další volnou pozici. Toto kopírování nenulových prvků a jejich sloupcových indexů do struktury CSR matice ukazuje výpis kódu 9.5. Je zřejmé, že toto řešení má operační složitost pouze $O(nnz(\mathbf{A}))$.

Paralelní varianty konverze COO matice do CSR formátu používají paralelní výpočet histogramu pro určení četností řádkových indexů. Paralelní výpočet histogramu nejprve rozdělí vstupní posloupnost dat na p bloků, kde p je počet vláken, a každé vlákno počítá svůj vlastní lokální histogram z bloku dat, který mu byl přidělen. Poté je rozsah četností rozdělen mezi p vláken a každé vlákno akumuluje data ze všech lokálních histogramů v rámci svého přiděleného rozsahu do výsledného globálního histogramu.

```

const std::size_t nnz = coo_mtx.m_vals.size();
std::vector<std::size_t> row_offset = csr_mtx.m_row_ptrs;

for (std::size_t i = 0; i < nnz; ++i) {
    const std::size_t row = coo_mtx.m_rows[i];
    const std::size_t dest = row_offset[row]++;
    csr_mtx.m_cols[dest] = coo_mtx.m_cols[i];
    csr_mtx.m_vals[dest] = coo_mtx.m_vals[i];
}

```

■ **Výpis kódu 9.5** Ukázka kopírování nenulových prvků a jejich sloupcových indexů z COO matice do CSR formátu v lineárním čase.

Jak bylo popsáno v kapitole 7.1.2, ani metoda přesné alokace nemůže předpovědět situace, kdy během akumulace dojde k algebraickému vynulování. Výstupní matice tedy může ve výjimečných případech obsahovat nulové prvky. V některých situacích však může uživatel požadovat, aby výstupní matice žádné nulové prvky vzniklé z algebraického vynulování při akumulaci neobsahovala. Je proto potřeba pro tyto účely implementovat třídu, která takovou matici převede na matici bez nenulových prvků. Tuto funkcionalitu implementuje metoda `prune_zeros` třídy `matrix_pruner`. Protože výstupem algoritmu násobení řídkých matic, který byl popsán v kapitole 8.1, je řídká matice ve formátu CSR, přijímá metoda `prune_zeros` jako parametr právě CSR matici.

Algoritmus převodu CSR matice na CSR matici bez nulových prvků nejprve vypočítá nové pole řádkových ukazatelů pomocí určení počtu nenulových prvků v každém řádku a následného prefixového součtu. Poté jsou pro jednotlivé řádky z původní matice zkopírovány nenulové prvky do výstupní matice na pozici, která je, obdobně jako u převodu COO matice na CSR matici, určená hodnotou z pole řádkových ukazatelů, přičemž tato hodnota je po zkopírování nenulového prvku inkrementována, aby ukazovala na další volnou pozici v polích pro sloupcové indexy a nenulové prvky výstupní matice.

Metoda `prune_zeros` je šablonová, neboť přijímá CSR matici typu `T`, kde šablonový parametr musí splňovat koncept `arithmetic` (viz. kapitola 9.3). Nabízí se tedy otázka, jakým způsobem porovnávat prvky typu `T` s nulou. Pro celočíselné datové typy je správným řešením prosté porovnání s nulou, nicméně pro datové typy s plovoucí řádovou čárkou by se mohlo zdát, že pro rozhodnutí o nulovosti prvku je vhodné porovnat tento prvek s relativní odchylkou `std::numeric_limits<T>::epsilon()`, která se nazývá strojové epsilon (z angl. *machine epsilon*). Tento přístup je však chybný, neboť strojové epsilon je rovno rozdílu mezi číslem 1,0 a nejbližší vyšší reprezentovatelnou hodnotou [62, 63]. Strojové epsilon tedy charakterizuje hustotu strojových čísel v okolí čísla 1, nikoliv v okolí nuly, přičemž strojová čísla mají v okolí nuly výrazně vyšší hustotu než v okolí čísla 1. Podle standardu IEEE-754 [71] je například v datovém typu s dvojitou přesností strojové epsilon rovno přibližně

$2,22 \cdot 10^{-16}$, zatímco nejmenší reprezentovatelná hodnota, která patří mezi tzv. subnormální strojová čísla, je rovna přibližně $4,94 \cdot 10^{-324}$. Nejmenší reprezentovatelná hodnota je tedy výrazně menší než strojové epsilon. Porovnání prvku se strojovým epsilon by tak mohlo značnou část validních nenulových prvků chybně označit za nulové a tento přístup je proto nepoužitelný. Strojové epsilon se používá v jiné situaci a to při rozhodování o rovnosti dvou prvků, kdy je strojové epsilon vztaženo k hodnotám porovnávaných prvků.

V případě odstranění nulových prvků z CSR matice je tedy nemožné rozhodnout, zda je daný nenulový prvek výsledkem zaokrouhlovací chyby, nebo zda se jedná o legitimní nenulovou hodnotu. Z těchto důvodů jsou v rámci metody `prune_zeros` všechny hodnoty, které nejsou reprezentovány přesnou nulou, považovány za legitimní nenulové prvky. Korektním řešením i v případě datových typů s plovoucí řádovou čárkou proto je porovnání s přesnou nulou, což je v této práci implementováno jako porovnání s hodnotou $T(0)$, kde T je datový typ nenulových prvků CSR matice. Je vhodné poznamenat, že tento postup funguje nejen pro celočíselné typy a typy s plovoucí řádovou čárkou, ale i pro komplexní datové typy `std::complex`.

9.7 Falešné sdílení a jeho eliminace

Jak bylo popsáno v kapitole 2.6, falešné sdílení může významně degradovat výkon paralelní aplikace, neboť může kvůli opakovanému zneplatňování bloků cache paměti jednotlivých výpočetních jader vést k částečné serializaci paralelního výpočtu. V některých situacích je falešné sdílení bohužel neodstranitelné, jako je tomu např. v případě kopírování nenulových prvků při převodu CSR matice s nulovými prvky z algebraického vynulování na CSR matici bez nulových prvků (viz. kapitola 9.6). Nicméně často se lze důmyslnými technikami falešnému sdílení zcela vyhnout.

Jedním ze způsobů eliminace falešného sdílení je tzv. jalová výplň (z angl. *dummy data*). Tato technika se používá v situacích, kdy všechna vlákna zapisují data do sdíleného pole o velikosti celkového počtu vláken, přičemž každé vlákno má v tomto poli vyhrazenou svou vlastní pozici pro zápis dat. Pokud jsou zapisované hodnoty primitivního datového typu, může se stát, že celé takové sdílené pole bude namapováno pouze do několika jednotek cache bloků. Je tedy zřejmé, že každý takový zápis způsobí zneplatnění cache bloku několika dalším výpočetním jádřům. Z těchto důvodů se datový typ zapisovaných dat uměle zvětšuje na násobek velikosti bloku cache paměti, čímž je zaručeno, že vlákno zápisem své hodnoty nezneplatní cache blok žádnému dalšímu výpočetnímu jádru.

Technika jalové výplně byla v tomto projektu použita při výpočtu paralelního prefixového součtu. Implementované řešení paralelního prefixového součtu, které pro paralelizaci nevyužívá externí knihovnu, používá algoritmus třířázkového paralelního prefixového součtu, který byl popsán v kapitole 9.5. V tomto algoritmu každé vlákno po výpočtu prefixového součtu svého bloku

```
template<typename T>
struct alignas(CACHE_LINE_SIZE) aligned_T {
    T val;
};
```

■ **Výpis kódu 9.6** Datová struktura zarovnaná na adresu dělitelnou velikostí bloku cache paměti.

dat zapisuje částečný součet tohoto bloku do sdíleného pole částečných součtů. Pro eliminaci falešného sdílení toto pole částečných součtů uchovává prvky datového typu `aligned_T`, který odpovídá struktuře zarovnané na počátek bloku cache paměti. Tato struktura jako členskou proměnnou obsahuje prvek typu `T`, který určuje datový typ prvků, nad kterými se vypočítává paralelní prefixový součet. Strukturu `aligned_T` ukazuje výpis kódu 9.6. Použití zarovnání `alignas(CACHE_LINE_SIZE)` přinutí kompilátor vyplnit velikost struktury tak, aby byla násobkem tohoto zarovnání.

Další technika odstranění falešného sdílení se používá u datového paralelismu při rozdělování iterací mezi výpočetní vlákna. Při nevhodném rozdělení iterací mezi výpočetní vlákna se může stát, že ačkoliv budou vlákna zapisovat data na různé pozice v rámci sdíleného pole, budou tyto pozice na adresách, které jsou sobě natolik blízké, že budou namapovány do stejného bloku cache paměti. V takovém případě dochází k falešnému sdílení a vlákna si vzájemně budou prostřednictvím koherenčního protokolu opakovaně zneplatňovat bloky cache paměti. Tomuto typu falešného sdílení se lze zcela vyhnout tím, že sdílené pole bude zarovnané na adrese dělitelné velikostí bloku cache paměti a jednotlivým výpočetním vláknům budou přidělovány bloky iterací tak, aby do jednoho bloku cache paměti zapisovalo vždy právě jedno vlákno.

Pro alokaci zarovnaného pole je možné použít speciální verzi alokační funkce `operator new`, která umožňuje specifikovat zarovnání prostřednictvím typu `std::align_val_t`. Tato varianta alokační funkce je k dispozici od C++17 a je významnou podporou eliminace falešného sdílení, neboť před C++17 neexistoval žádný přenositelný způsob pro alokaci zarovnaného pole `dat`. Takto alokované pole je pak nezbytné uvolnit analogickou variantou dealokační funkce `operator delete`, která kromě ukazatele přijímá i specifikaci zarovnání.

Ruční volání speciální dealokační funkce pro každé pole, které bylo alokováno se zarovnáním, je náchylné k chybám. Z toho důvodu je pro účely udržitelnosti kódu v implementovaném řešení alokované pole spravováno prostřednictvím chytrého ukazatele, kterému je jako šablonový parametr předán funktor, který tuto speciální variantu dealokačního operátoru volá.

Pro určení velikosti jednoho bloku cache paměti byla použita hodnota `std::hardware_destructive_interference_size`, která je definována jako minimální vzdálenost mezi dvěma objekty nutná k zabránění falešnému sdílení. [62, 63] Tato hodnota je dostupná od verze C++17, nicméně některé starší překladače, jako je například GCC 11, který byl použit na systému pro vývoj

v rámci této práce, tuto konstantu neimplementují. Proto je v implementovaném řešení použit blok podmíněného překladu, který v případě, že tato konstanta není překladačem implementována, použije ručně definovanou hodnotu 64 bajtů.

9.8 Předávání výsledků hodnotou a jejich optimalizace

V implementovaném řešení je často výsledek výpočtu z jednotlivých metod vracen prostřednictvím návratové hodnoty. Tento přístup je pro jazyk C++ trochu atypický, neboť na rozdíl od jiných programovacích jazyků, jako je například Java nebo Python, které objekty předávají odkazem, jsou v jazyce C++ objekty standardně předávány hodnotou. Způsob předávání hodnotou má za následek, že například při přiřazení dočasného objektu do proměnné stejného typu se může vytvořit kopie celého přiřazovaného objektu, což může být výpočetně velmi neefektivní, obzvláště pro datově náročné struktury, jako jsou např. COO nebo CSR matice. Nicméně moderní standardy C++ zavedly mechanismy, které kompilátoru dovolují či přímo zavazují eliminovat zbytečné kopie a přesuny objektů. Mezi tyto mechanismy patří optimalizace pojmenované návratové hodnoty (z angl. *named return value optimization*, zkr. NRVO) a tzv. odložené vytvoření dočasného objektu (z angl. *deferred temporary materialization*).

NRVO je optimalizace, při níž kompilátor rozpozná, že pojmenovaná lokální proměnná funkce či metody bude použita jako návratová hodnota, a místo jejího zkopírování do návratové hodnoty ji rovnou sestaví na paměťovém místě, kde bude výsledek uložen na straně volajícího. Kompilátor tedy sloučí paměťové umístění lokální proměnné s paměťovým umístěním návratové hodnoty, čímž eliminuje jak destrukci lokálního objektu, tak potenciální přesun nebo kopii při návratu z funkce či metody. Ukázku NRVO je možné vidět ve výpise kódu 9.7. Bez NRVO by kompilátor při příkazu `return csr_mtx;` zkopíroval nebo přesunul objekt `csr_mtx` do paměti vyhrazené pro návratovou hodnotu a poté by na lokální objekt `csr_mtx` zavolal jeho destruktorku. Při použití NRVO kompilátor od počátku sestaví `csr_mtx` přímo v paměťové oblasti určené pro návratovou hodnotu, takže žádná kopie ani přesun vůbec nevzniknou. NRVO je speciálním případem optimalizace, která se nazývá *copy elision* a je v C++ standardu definována jako volitelná optimalizace od C++11. Nicméně kompilátory GCC i Clang tuto optimalizaci automaticky aplikují, a to dokonce i při explicitně vypnutých optimalizacích přepínačem `-O0`.

Odložené vytvoření dočasného objektu bylo zavedeno v C++17 a tento mechanismus zaručuje, že vytvoření dočasného objektu v paměti bude odloženo do okamžiku, kdy je objekt skutečně potřebný. Pokud je např. dočasným objektem inicializována proměnná, jako je tomu ve výpise 9.8, pak kompilátor tento objekt sestaví přímo v paměti dané proměnné bez jakéhokoliv kopíro-

```
template<arithmetic T>
csr_matrix<T> seq_coo_to_csr::convert(
    const coo_matrix<T> & coo_mtx) const {
    csr_matrix<T> csr_mtx;
    //...
    return csr_mtx;
}
```

■ **Výpis kódu 9.7** Optimalizace NRVO na příkladu sekvenční konverze COO matice na CSR matici.

```
csr_matrix<T> mtx_a_csr = m_coo_to_csr.convert(mtx_a_coo);
csr_matrix<T> mtx_b_csr = m_coo_to_csr.convert(mtx_b_coo);
```

■ **Výpis kódu 9.8** Ukázka aplikace odloženého vytvoření dočasného objektu.

vaní či přesunu obsahu. Pro ukázku 9.8 to tedy znamená, že dočasné objekty z metody `convert` jsou zkonstruovány přímo v paměťovém prostoru proměnných `mtx_a_csr` a `mtx_b_csr`, a není tedy volán kopírovací ani přesouvací konstruktor. Tento mechanismus bude dokonce aplikován i v případech, kdy jsou kopírovací a přesouvací konstruktory explicitně zakázány. Na rozdíl od NRVO není odložené vytvoření dočasného objektu volitelným mechanismem a všechny kompilátory jej musí povinně aplikovat.

Díky optimalizaci návratové hodnoty a odloženému vytvoření dočasného objektu je proto možné přiřazovat výsledky výpočtů prostřednictvím dočasného objektu, aniž by tento způsob přinášel jakoukoliv dodatečnou režii. Tento přístup navíc výrazně zlepšuje čitelnost kódu oproti přístupu, kdy by byl výsledný objekt metod předáván jako výstupní parametr prostřednictvím reference nebo chytrého ukazatele.

9.9 CLI aplikace a její rozhraní

Všechny tři varianty násobení řídkých matic byly implementovány jako CLI aplikace, aby bylo možné aplikace spouštět prostřednictvím vzdáleného přístupu na výpočetních klastrech, jako je například fakultní klastr ClusterFIT. Detailní informace o používání těchto aplikací je možné nalézt v uživatelské příručce v příloze A. Veškeré argumenty budou tedy aplikacím předávány z příkazové řádky. Pro implementaci zpracování argumentů z příkazové řádky byla použita externí knihovna `argparse` [72]. Logika zpracování argumentů je zapouzdřena do třídy `argument_parser`, která funguje jako prostředník mezi parametry `argc` a `argv` poskytovanými operačním systémem a konfigurací SpGEMM výpočtu reprezentovanou strukturou `spgemm_setup`, se kterou pracují třídy jednotlivých aplikací.

Knihovna `argparse` [72] rozděluje argumenty příkazové řádky do dvou hlavních skupin. První skupinou jsou poziční argumenty, které jsou povinné a záleží na jejich pořadí. Všechny tři varianty implementované aplikace používají dva poziční argumenty pro specifikaci cest k souborům matic **A** a **B** ve formátu Matrix Market.

Druhou skupinou argumentů jsou volitelné argumenty, které mají podobu krátkých a dlouhých přepínačů. V implementaci jsou přepínače použity pro specifikaci argumentů, které nastavují parametry samotného SpGEMM výpočtu, přičemž všem přepínačům byly nastaveny výchozí hodnoty. Pomocí volitelných argumentů lze určit, jestli mají být ve výstupní matici **C** seřazeny nenulové prvky v každém jejím řádku podle sloupcového indexu, nebo zda mají být z této matice odstraněny nenulové prvky, které mohly vzniknout algebraickým vynulováním. Dále volitelné argumenty umožňují nastavit, zda má být výsledná matice **C** zapsána ve formátu Matrix Market, nebo zda mají být na výstup zapsána neformátovaná data ze struktury `csr_matrix<T>`. Dále je možné přepínačem specifikovat, zda má být výsledná matice **C** zapsána na standardní výstup, nebo zda má být zapsána do souboru. Pro paralelní varianty je dále možné určit počet vláken, které se mají účastnit výpočtu, přičemž pokud není tento počet specifikován uživatelem, je jako výchozí hodnota použit výraz `std::thread::hardware_concurrency`. Tento výraz však může být nulový v situacích, kdy hodnota podporovaného počtu souběžných vláken není definována nebo ji nelze vypočítat. [62, 63] Proto je v případě, kdy není hodnota počtu vláken zadána explicitně uživatelem a výraz `std::thread::hardware_concurrency` je nulový, z výše uvedených důvodů použita hodnota 1.

Knihovna `argparse` dále umožňuje pro všechny argumenty specifikovat nápovědu. Tato data jsou pak použita pro automatické generování nápovědy programu, kterou lze získat pomocí přepínače `--help`, takže není nutné nápovědu programu prostřednictvím tohoto přepínače specifikovat ručně.

Po zpracování argumentů z příkazové řádky a nastavení datové struktury `spgemm_setup` je spuštěn samotný výpočet. SpGEMM výpočet se skládá z několika na sebe navazujících kroků, kterými jsou načtení matic ze souborů, konverze z formátu COO do formátu CSR, samotné násobení matic v tomto formátu, volitelné odstranění nulových prvků a zápis výsledku do souboru nebo na standardní výstup. Umístění veškerého tohoto řídicího kódu do hlavní funkce `main` by vedlo ke špatně čitelnému a obtížně rozšiřitelnému kódu. Z toho důvodu byl veškerý řídicí kód zapouzdřen do samostatné třídy prostřednictvím návrhového vzoru fasáda (z angl. *facade*).

Fasáda je návrhový vzor popsáný v knize [73], a jeho cílem je poskytnout zjednodušené a ucelené rozhraní ke složitějšímu subsystému, který zahrnuje použití většího množství tříd a zajišťuje správnou interakci mezi těmito třídami. Volající pak komunikuje výhradně s fasádou a není nucen znát vnitřní strukturu tohoto subsystému. [73] V kontextu této práce fasáda agreguje na-

čtení vstupu, konverzi formátu, násobení řídkých matic, odstranění nulových prvků z algebraického vynulování a zápis výsledku do jedné metody `run`.

Jak již bylo popsáno v kapitole 9.3, jedním z hlavních přínosů této implementace je, že dokáže načíst jakoukoliv matici uloženou v korektním Matrix Market formátu. Další výjimečnou vlastností je, že datový typ nenulových prvků načtené matice je určen dynamicky na základě informace v hlavičce Matrix Market souboru. Pokud je tedy například v hlavičce Matrix Market souboru uveden datový typ `real`, bude matice načtena do struktury `coo_matrix<double>`. Ačkoliv mají v případě SpGEMM násobení vstupní matice **A** i **B** většinou stejný datový typ, je nezbytné ošetřit i případ, kdy budou rozdílného datového typu. V takovém případě se určí společný datový typ obou matic jako sjednocení těchto datových typů. Pokud je tedy matice **A** datového typu `integer` a matice **B** datového typu `complex`, pak jsou obě matice načteny do struktur `coo_matrix<std::complex<double>>`. Na základě určeného společného datového typu je pak zavolána šablonová metoda `run_spgemm<T>`, která vykoná celou posloupnost operací popsanou výše.

V rámci této posloupnosti operací je zároveň detekováno, jestli jsou cesty a názvy souborů matic **A** a **B** různé. Pokud jsou jako argumenty CLI aplikace zadány dva stejné soubory, což reprezentuje výpočet $\mathbf{C} = \mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$, je vstupní matice **A** načtena pouze jednou. Metodě `multiply` pro násobení dvou řídkých matic je pak pro levou i pravou matici předána reference na tentýž objekt. Díky této optimalizaci nebude při výpočtech typu $\mathbf{A}^2$ docházet k plýtvání pamětí a navíc nebude docházet k redundantnímu načítání dat, což příznivě ovlivňuje efektivitu využití cache paměti, neboť procesor pracuje stále se stejným objektem namísto dvou nezávislých kopií matice **A**.

Testování, automatizace sestavení a kontinuální integrace

Tato kapitola popisuje C++ framework použitý pro testování a popisuje způsob testování implementovaných tříd. Dále se je popsán způsob automatizace sestavení projektu včetně automatického vygenerování dokumentace. Závěr kapitoly se zabývá popisem nastavení kontinuální integrace na fakultní instanci repozitáře GitLab.

10.1 Testování

Pro účely testování implementovaných tříd byl zvolen testovací framework Doctest [74]. Tento testovací framework podporuje C++20, disponuje velmi snadnou integrací a navíc umožňuje používat testovací makra i v běhu s více vlákny, která jsou vytvořena v rámci jednoho testovacího případu. Doctest je inspirován testovacím frameworkem Catch2 [75], odlišuje se však výrazně kratší dobou kompilace a vyšší rychlostí provedení testů. Další výhodou tohoto frameworku je, že je distribuován jako knihovna s jedním hlavičkovým souborem (z angl. *single-header library*), nevyžaduje žádné externí závislosti ani složité konfigurace sestavení. [74]

Pro integraci tohoto testovacího frameworku stačí pouze zahrnout hlavičkový soubor `doctest.h` a pro každý spustitelný testovací soubor zajistit, aby v právě jedné jeho jednotce překladu byl definován spouštěč testů prostřednictvím makra `DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN`. Toto makro zajistí automatické vygenerování hlavní vstupní funkce `main` včetně automatického vyhledání, zaregistrování a spuštění všech testovacích případů. Všechny ostatní jednotky překladu ve stejném spustitelném souboru, které také používají framework Doctest, pak pouze zahrnují hlavičku tohoto frameworku bez výše

uvedeného makra, aby bylo možné i tyto jednotky překladu zapojit do registru testů, avšak bez vytváření dalších funkcí `main`. Výstupem kompilace je CLI aplikace, která umožňuje spuštění konkrétních nebo všech testovacích případů v rámci daného souboru. [74]

Testovací framework Doctest používá syntaxi založenou na makrech. Mezi základní makra, která jsou používána v rámci této práce, patří následující makra:

- **TEST_CASE**: Testy jsou organizovány do testovacích případů prostřednictvím makra `TEST_CASE`. Toto makro přijímá řetězec, který slouží jako jednoznačný identifikátor daného testovacího případu. [74]
- **SUBCASE**: V rámci jednoho testovacího případu je možné související testovací scénáře seskupit do podpřípadů prostřednictvím makra `SUBCASE`. Pokud testovací případ obsahuje několik testovacích podpřípadů, pak je pro každý podpřípad spuštěn daný `TEST_CASE` samostatně od začátku. Z toho plyne, že veškerý kód, který je napsán v těle testovacího případu před makry `SUBCASE`, plní funkci sdílené společné inicializace, která se spustí pro každý podpřípad samostatně. Tato vlastnost je užitečná pro seskupování souvisejících, ale nezávislých scénářů, které sdílejí společné počáteční nastavení. [74]
- **CHECK**: Základním makrem pro aserce je makro `CHECK`. Pokud výraz, který je tomu makru předán, je vyhodnocen jako nepravdivý, pak je tato situace vyhodnocena jako selhání aserce, avšak pokračuje se v provádění zbytku testovacího případu. Tento přístup umožňuje identifikovat všechny chyby v rámci jednoho testovacího běhu. [74]
- **CHECK_FALSE**: Použití tohoto makra je ekvivalentní použití makra `CHECK` s negovaným výrazem. V rámci této práce je toto makro použito pro lepší čitelnost při ověření, že daná podmínka není splněna.
- **REQUIRE**: Toto makro se používá pro kritické aserce. Na rozdíl od makra `CHECK` je v případě selhání testovací podmínky okamžitě ukončen aktuální testovací případ. V rámci této práce je makro `REQUIRE` použito pro ověření podmínek, jejichž nesplnění by způsobilo, že by následující ověření postrádala smysl nebo rovnou vedla k selhání testovacího programu. [74]
- **CHECK_THROWS_AS**: Toto makro se používá k ověření, že poskytnutý výraz vyvolá výjimku zadaného typu. V rámci této práce slouží k ověření, že jsou neplatné vstupy odmítnuty se správnou výjimkou. [74]

Pokud některá aserce selže, framework Doctest vypíše informaci obsahující název zdrojového testovacího souboru, číslo řádku a vyhodnocený výraz včetně konkrétních hodnot jeho dílčích výrazů. Tento přístup umožňuje snadnou diagnostiku testovacích případů, jejichž aserce selhaly, a následnou identifikaci problému. [74]

```
class argv_builder {  
private:  
    std::vector<std::string> m_strings;  
    std::vector<char*> m_ptrs;  
public:  
    argv_builder(  
        const std::initializer_list<std::string> & args);  
  
    int argc() const;  
  
    char ** argv();  
};
```

■ **Výpis kódu 10.1** Třída pro generování argumentů způsobem, jakým je operační systém předává programu.

Testování v rámci této práce bylo realizováno prostřednictvím jednotkových testů (z angl. *unit tests*). Testy jsou strukturovány jako samostatné spustitelné soubory pro každou testovanou třídu, tj. každý spustitelný testovací soubor pokrývá právě jednu třídu. Datové struktury, které slouží pouze k agregaci dat a neobsahují žádnou logiku (jako např. `coo_matrix`), jsou z testování vynechány, neboť jejich testování by nepřinášelo žádnou hodnotu, protože by byla testována pouze správnost C++ překladače. Navíc je korektnost těchto datových struktur otestována nepřímo v rámci jednotkových testů tříd, které je používají jako vstupní nebo výstupní reprezentaci dat.

Ve všech testovacích souborech jsou testovací případy strukturovány obdobným způsobem. Ověřena je vždy základní konstrukce objektu, jeho počáteční stav a destrukce. Následně je ověřena funkcionálnost veřejného rozhraní třídy, dále jsou otestovány mezní hodnoty a také chybové stavy pomocí ověření vyvolání správné výjimky. Na závěr je funkcionálnost třídy prověřena zevrubným testem náhodnými daty. Testy náhodnými daty používají jako generátor pseudonáhodných čísel objekt typu `std::mt19937`, který implementuje algoritmus Mersenne Twister pro 32-bitová data. [62, 63] Pro zajištění reprodukovatelnosti testů náhodnými daty je tento generátor pseudonáhodných čísel inicializován konstantní hodnotou.

Ačkoliv jsou testovací soubory organizovány podobně, některé implementační detaily stojí alespoň za krátkou zmínku. Konstruktor třídy pro zpracování CLI argumentů `argument_parser` (viz. kapitola 9.9) přijímá stejné parametry jako hlavní funkce programu `main`, neboť tyto parametry musí být předány konstruktoru třídy z externí knihovny `arg_parse`. V rámci testování bylo proto nezbytné navrhnout způsob, jakým bude možné vytvořit argumenty způsobem, jakým je předává operační systém hlavní funkci programu, tj. vytvořit ukazatel na pole řetězců z jazyka C, jehož prvky reprezentují argumenty z příkazové řádky. Pro tyto účely byla implementována pomocná třída `argv_builder`, jejíž signaturu ukazuje výpis kódu 10.1.

```
argv_builder args{"/spgemm",
                  "matrices/matrix_a.mtx",
                  "../matrix_b.mtx"};

argument_parser parser(args argc(), args.argv());
```

■ **Výpis kódu 10.2** Ukázka použití třídy `argv_builder`.

Konstruktor třídy `argv_builder` přijímá konstantní referenci na objekt typu `std::initializer_list`, který představuje nemodifikovatelnou posloupnost prvků stejného typu. Tento datový typ je překladačem automaticky vytvořen v situacích, kdy je použit seznam inicializátorů, tj. posloupnost prvků stejného typu uzavřená ve složených závorkách, na místě, kde cílový typ tento seznam přijímá jako argument. Příklad použití této techniky ukazuje výpis kódu 10.2. Instance třídy `argv_builder` je inicializována seznamem inicializátorů, který obsahuje tři literály typu `std::string`, z nichž kompilátor automaticky vytvoří dočasný objekt typu `std::initializer_list<std::string>`, na nějž je navázána konstantní reference konstruktoru třídy `argv_builder`. Tímto objektem je v konstruktoru následně inicializována proměnná `m_strings`, neboť jeden z konstruktorů třídy `std::vector` jako parametr přijímá právě `std::initializer_list`.

Dále stojí za zmínku testování třídy `flat_hash_map`. Pro tuto třídu bylo nezbytné otestovat korektní zápis a přepisování náhrobků. K tomuto účelu bylo potřeba zajistit vygenerování stejné vyhledávací posloupnosti. Jak bylo popsáno v kapitole 9.1, implementovaná hešovací tabulka zahešuje klíč dvakrát. Nejprve je klíč zahešován hešovací funkcí z šablonového parametru a následně je tento výstup zahešován interní hešovací funkcí vyšších bitů součinu s multiplikativní konstantou podle Donalda Knutha. Díky tomuto důmyslnému návrhu bylo možné testování náhrobků realizovat tak, že vytvořené instanci je předána hešovací funkce, která pro každý vstup vrací stejný heš. Tím je zaručeno, že vyhledávací posloupnost bude pro všechny prvky totožná.

U tříd pro konverzi COO matice na CSR matici bylo potřeba ověřit rovnost dvou objektů typu `std::vector`, které sice obsahují stejné prvky, ale je možné, že se tyto prvky nacházejí na různých pozicích. Jelikož standardní operátor rovnosti ověřuje, zda vektory obsahují stejné prvky na stejných pozicích, bylo nezbytné zvolit jiný přístup. Pro tento účel byla zvolena knihovní funkce `std::is_permutation`, která ověřuje, zda daný rozsah specifikovaný iterátory je permutací druhého rozsahu. Složitost této funkce je $O(n^2)$, kde n označuje velikost vstupních rozsahů. Řešení, které oba vstupní vektory nejprve seřadí a následně je porovná operátorem rovnosti, je sice rychlejší, neboť má složitost $O(n + n \cdot \log n)$, avšak tento přístup kvůli seřazení musí modifikovat oba vstupní vektory, případně musí oba vektory zkopírovat do dočasných proměnných, které jsou následně seřazeny a porovnány. Z toho důvodu a kvůli výrazně lepší čitelnosti kódu bylo zvoleno řešení založené na funkci `std::is_permutation`.

10.2 Automatizace sestavení

Pro automatizaci sestavení, zpracování externích knihoven, vyhledání testů a vygenerování dokumentace byl zvolen nástroj CMake [76]. Na rozdíl od nástrojů pro automatizaci překladu, jako je např. Make [77] nebo Ninja [78], však CMake místo přímé kompilace zdrojových kódů generuje vstupní soubory určené konkrétnímu nástroji pro automatizaci překladu. Pomocí nástroje CMake je tedy možné vygenerovat soubory pro sestavení určené pro nástroj Make, Ninja nebo jiný nástroj pro automatizaci překladu. Konkrétní konfigurace sestavení je definována prostřednictvím jednoho nebo více souborů **CMakeLists**, na základě kterých CMake vygeneruje soubory pro sestavení, které jsou uzpůsobeny konkrétnímu nástroji pro automatizaci překladu.

Významnou výhodou nástroje CMake je schopnost automaticky stahovat externí knihovny pomocí modulu **FetchContent**. Pomocí tohoto modulu jsou externí knihovny staženy automaticky při generování souborů pro sestavení a zároveň je zajištěno, že stažené knihovny budou automaticky zahrnuty při překladu. Díky této funkcionalitě odpadá nutnost ruční instalace externích závislostí na každém systému, na kterém má být spuštěna kompilace, a navíc je zajištěno, že všechna vývojová i testovací prostředí používají stejné verze externích knihoven. V rámci této práce byla tato funkcionalita využita pro všechny externí knihovny, tj. `argparse`, `fast_matrix_market` a `Doctest`.

Přenositelným způsobem je možné spravovat i systémové knihovny. Paralelní varianty využívající OpenMP a C++ vlákna potřebují nalinkovat potřebné knihovny pro vícevláknové programování, nicméně místo pevně zadaných přepínačů pro překladač umožňuje CMake spravovat systémové knihovny prostřednictvím funkce **find_package**. Pomocí této funkce je možné specifikovat knihovnu a nalinkovat ji pro danou cílovou aplikaci přenositelným způsobem. Například varianta SpGEMM násobení využívající C++ vlákna potřebuje při překladu na Unixových operačních systémech zahrnout knihovnu Pthreads (viz. kapitola 3), zatímco na operačních systémech Windows není potřeba žádný přepínač, neboť knihovna Windows Threading Library je zahrnuta automaticky. Nástroj CMake systémovou knihovnu pro vícevláknové programování abstrahuje prostřednictvím balíčku **Threads** a nastavení potřebného přepínače pak probíhá automaticky na základě operačního systému, na kterém je spuštěno sestavení.

CMake dále umožňuje pro automatizaci testování použít nástroj CTest. Jak bylo popsáno v kapitole 10.1, každý testovací soubor je definován jako samostatný spustitelný soubor. Funkce `doctest_discover_tests` z frameworku Doctest automaticky zjišťuje u každého testovacího binárního souboru, jaké testovací případy obsahuje, a automaticky je registruje v nástroji CTest. Díky tomu přidání nového testovacího případu do zdrojového souboru nevyžaduje žádné změny v konfiguraci, neboť tento testovací případ bude detekován a zaregistrován automaticky. Veškeré testy lze po kompilaci spustit jednotně prostřednictvím příkazu `ctest`.

Pro generování dokumentace je použit nástroj Doxygen, který extrahuje dokumentaci z anotací zdrojového kódu a generuje ji ve formátu HTML. Tato závislost byla zvolena jako volitelná, tj. pokud je v cílovém systému nalezen nástroj Doxygen, vytvoří se cíl pro sestavení dokumentace. Pokud na cílovém systému není Doxygen k dispozici, konfigurace pokračuje bez chyby a cíl pro sestavení dokumentace se nevytvoří, přičemž všechny ostatní cíle sestavení zůstanou neovlivněny. Jako volitelná byla nastavena i závislost na nástroji Graphviz, pomocí kterého jsou automaticky generovány diagramy znázorňující dědičnost tříd.

10.3 Kontinuální integrace

V moderním softwarovém inženýrství představuje kontinuální integrace (z angl. *continuous integration*, zkr. CI) postup, ve kterém je pro každou změnu zdrojového kódu, která je zapsaná do verzovacího systému, spuštěna posloupnost předem definovaných automatizovaných procesů. Zdrojový kód aplikací implementovaných v rámci této diplomové práce byl verzován v systému Git a jako webový repozitář byla zvolena fakultní instance systému GitLab. Pro kontinuální integraci byl použit GitLab CI, který byl využit pro automatizaci kompilace, spuštění a ověření kompletní sady implementovaných testů a generování dokumentace. Posloupnost těchto tří procesů je spuštěna automaticky při každém odeslání nových změn do webového repozitáře. Kompletní konfigurace kontinuální integrace je popsána v souboru `.gitlab-ci.yml` v kořenovém adresáři projektu.

Konkrétní spuštění sady automatizovaných procesů se často označuje jako *pipeline*. V GitLab CI se pipeline skládá z úloh (v angl. *jobs*), které určují, jaké konkrétní úkoly se mají vykonat (jako je např. kompilace zdrojového kódu), a fází (z angl. *stages*), které určují, kdy se úlohy mají spustit. Fáze jsou spuštěny postupně v pořadí deklarace v konfiguračním souboru, přičemž úlohy v rámci jedné fáze mohou být spuštěny paralelně. V rámci tohoto projektu byly použity tři fáze, každá s právě jednou úlohou. Fáze `build` obstarává překlad zdrojových kódů, fáze `test` spouští a ověřuje všechny implementované testy a fáze `deploy` generuje dokumentaci a nahrává ji na PagesFIT (viz. dále).

Každá úloha v rámci GitLab CI pipeline je spuštěna v izolovaném prostředí kontejneru Docker. Technologie Docker slouží pro vytváření a spuštění kontejnerů, které představují izolované procesy, jež sdílejí jádro hostitelského operačního systému, avšak mají svůj vlastní oddělený jmenný prostor pro procesy a souborový systém. Docker obraz (z angl. *image*) je nemodifikovatelný záznam souborového systému, který slouží jako šablona, ze které jsou kontejnery instanciovány. Tyto obrazy se distribuují prostřednictvím registrů, z nichž nejpoužívanější je Docker Hub [79].

Když je v GitLab CI spuštěna úloha, je z registru stažen obraz, který je uveden v konfiguračním souboru. Ze staženého obrazu je vytvořen kontejner

a příkazy dané úlohy jsou spuštěny v rámci tohoto kontejneru. Po dokončení úlohy je kontejner smazán a pro další úlohu se musí vytvořit nový kontejner. Pro účely této práce byl jako Docker obraz zvolen oficiální obraz překladače GCC ve verzi 13.4.0, neboť na fakultním klastru, na kterém bude prováděno měření, je k dispozici právě překladač GCC 13.

Vzhledem k tomu, že se každá úloha v rámci pipeline spouští v izolovaném kontejneru, je toto prostředí bezstavové a soubory vytvořené v rámci jedné úlohy se do další úlohy nepřenášejí. K ukládání souborů, které jsou vytvořeny v rámci jedné úlohy, slouží artefakty (z ang. *artifacts*). Artefakty jsou soubory a složky, které jsou na konci úlohy nahrány na GitLab server a jsou k dispozici ke stažení pro následující úlohy v rámci stejné pipeline nebo pro uživatele prostřednictvím webového rozhraní. V rámci tohoto projektu byl jako artefakt definován celý adresář, ve kterém proběhla kompilace, neboť je nezbytný pro fáze **test** a **deploy**. Jako druhý artefakt byl nastaven výstup generování dokumentace. Aby nebyla překročena kvóta úložiště projektu, bylo v konfiguračním souboru pro oba artefakty nastaveno, aby byly po 2 hodinách od nahrání automaticky smazány. Pro artefakty z nejnovější pipeline byla ve webovém repozitáři GitLab nastavena výjimka, aby nebyly smazány, dokud nebudou nahrány novější artefakty.

Poslední fáze **deploy** je kromě vygenerování dokumentace v podobě HTML souborů také odpovědná za jejich automatické nasazení. Úloha pro generování dokumentace byla pojmenována jako **pages**, což je vyhrazený název v GitLab CI pro úlohu, jejíž artefakt, pokud je pojmenován jako **public**, je automaticky zpřístupněn v podobě statických webových stránek. Konkrétní URL adresa, na které je takto automaticky vygenerovaná a nasazená dokumentace dostupná, určuje na fakultní instanci systému GitLab služba PagesFIT. Pro správnou funkcionalitu této služby je také nezbytné v konfiguračním souboru `.fit-pages.yml` specifikovat uživatele, kteří mají mít k těmto webovým stránkám přístup. Více o službě PagesFIT je možné se dočíst v manuálových stránkách [80]. Dokumentace tohoto projektu je veřejně dostupná na adrese `javormat.pages.fit/ni-dip-spgemm` a v konfiguračním souboru `.fit-pages.yml` byl přístup k těmto stránkám povolen všem uživatelům, včetně anonymních návštěvníků.

Měření a vyhodnocení

Tato kapitola se věnuje popisu konfigurace fakultního klastru ClusterFIT, na kterém bylo provedeno měření. Dále je popsána aplikace, která byla pro účely měření implementována, a také způsob měření včetně volby vhodných matic. Na závěr jsou popsány a diskutovány výsledky měření.

11.1 Specifikace cílového systému

K řešení náročných výpočetních úloh je na fakultě k dispozici výpočetní klastř ClusterFIT. Tento klastř disponuje několika desítkami výpočetních uzlů, avšak většina těchto uzlů je virtualizovaných, což může negativně ovlivnit výkonnost výpočtu. Aby měly naměřené výsledky co nejvíce vypovídající hodnotu, byly pro měření zvoleny uzly, na kterých je v rámci fakultního klastru přímý přístup k hardwaru bez jakékoliv virtualizační vrstvy. Mezi nevirtualizované uzly patří uzly s procesory Fujitsu A64FX založené na CPU architektuře AArch64 (známé také jako ARM64) a uzel s procesorem AMD EPYC 7543P používající architekturu x86_64 (někdy označovanou také jako AMD64). Vzhledem k tomu, že architektura AArch64 je oproti ostatním mírně specifická, neboť například oproti architektuře x86_64 používá redukovanou sadu instrukcí, bude měření pro účely porovnání provedeno jak na výpočetních uzlech s procesory Fujitsu A64FX, tak i na uzlu s procesorem AMD EPYC 7543P.

Výpočetní uzly jsou používány primárně v neinteraktivním režimu a jsou přístupné přes plánovač úloh, který zajišťuje vyvážené využití dostupných výpočetních prostředků. Pro interaktivní přístup je vyhrazen tzv. frontend uzel, který slouží k překládání zdrojových kódů a k plánování úloh prostřednictvím plánovače Slurm. Každá úloha pro měření proto musí být zařazena do některé fronty úloh. Pro výpočetní uzly s procesory Fujitsu A64FX byla použita fronta `arm_vip`, ve které může jedna úloha provádět svůj výpočet po dobu až 24 hodin. Pro výpočetní uzel s procesorem AMD EPYC 7543P je vyhrazena fronta `amd` s maximální dobou výpočtu jedné úlohy 4 hodiny. Pokud v rámci měření některá úloha překročí maximální povolený časový limit dané fronty

úloh, bude ve výsledcích pro danou matici uveden symbol uzavírací lomené závorky a časový limit této fronty, tj. například při překročení časového limitu ve frontě `amd` bude ve výsledcích uveden údaj > 4 h.

Na ClusterFIT má každý výpočetní uzel s procesorem Fujitsu A64FX uvnitř CPU 4 paměťové uzly typu NUMA, z nichž každý má 12 výpočetních jader. Jeden uzel má proto k dispozici celkem 48 výpočetních jader. Každý uzel disponuje 32 GB operační paměti a jako operační systém používá Rocky-Linux 8. [9]

Procesor AMD EPYC 7543P disponuje 32 fyzickými jádry, avšak každé jádro je schopné vykonávat dva nezávislé proudy instrukcí. Jeden výpočetní uzel s procesorem AMD EPYC 7543P má proto k dispozici 64 logických jader. Každý takový uzel má 512 GB operační paměti a používá Ubuntu 24.04 LTS jako operační systém. [81]

11.2 Způsob měření

Matice pro měření budou vybírány z kolekce SuiteSparse Matrix Collection (dříve známé pod názvem The University of Florida Sparse Matrix Collection), neboť jde o největší veřejně dostupnou kolekci řídkých matic. [24] Tato kolekce také obsahuje většinu matic z dříve publikovaných kolekcí, jako jsou Sparse Matrix Test Problems [41], A Web Resource for Test Matrix Collections [42] a The RutherfordBoeing Sparse Matrix Collection [43]. Kolekce SuiteSparse Matrix Collection se navíc nadále rozrůstá a v době psaní této práce obsahovala více než 2900 řídkých matic. Celá kolekce je veřejně dostupná na webové adrese <https://sparse.tamu.edu/>.

Většina matic v této kolekci pochází z reálných problémů a aplikací, proto většina z nich má rozdílné rozměry. Nalézt páry řídkých matic, které mají kompatibilní rozměry pro násobení řídkých matic, je tedy velice obtížné. Nicméně značná většina matic v této kolekci jsou čtvercové matice, konkrétně ze všech 2904 matic je 2226 matic čtvercových. Pro účely měření proto budou vybrány čtvercové matice a bude změřen čas výpočtu $\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$, kde $\mathbf{A} \in \mathbb{C}^{n,n}$ je vstupní čtvercová matice.

Nejmenší čtvercová matice v SuiteSparse Matrix Collection má řád 2, největší čtvercová matice má řád 226 196 185. Aby bylo pokryto co nejširší spektrum řídkých matic, budou matice vybrané pro účely měření rozděleny do 5 kategorií podle rozměrů. Tyto kategorie velikostí matic pro měření ukazuje tabulka 11.1. Z každé kategorie bude vybrána alespoň jedna matice s celočíselnými (datové typy `pattern` a `integer`), reálnými (datový typ `real`) a komplexními prvky (datový typ `complex`). Výjimkou jsou poslední dvě kategorie, pro které není v SuiteSparse Matrix Collection [24] dostupná žádná čtvercová řídká matice s komplexními nenulovými prvky. Popis řídkých matic, které byly pro jednotlivé skupiny vybrány, je možné nalézt v příloze B.

Celkem bylo pro měření vybráno 42 matic, a ačkoliv jsou všechny matice volně dostupné ke stažení přímo ze stránek kolekce SuiteSparse Matrix

Collection [24], byly pro snazší manipulaci a pro účely této práce všechny vybrané matice rozděleny do složek podle kategorií popsanych výše a zabaleny do jednoho archivu. Tento archiv byl nahrán na datové úložiště ownCloud [82] pro sdílení akademických dat, které provozuje sdružení CESNET. Nahráný archiv je veřejně dostupný na odkazu <https://owncloud.cesnet.cz/index.php/s/GaK9eHieKEv8CB9>.

Skupina čtvercových řídkých matic	Řád čtvercové matice
Velmi malé	[2; 10 000)
Malé	[10 000; 100 000)
Střední	[100 000; 1 000 000)
Velké	[1 000 000; 10 000 000)
Velmi velké	[10 000 000; 226 196 185]

■ **Tabulka 11.1** Rozdělení matic pro měření podle řádu čtvercové matice.

Pro účely měření byla implementována samostatná aplikace `measurements`. Původním záměrem bylo implementovat měření tak, že pro vstupní čtvercovou matici $\mathbf{A} \in \mathbb{C}^{n,n}$ bude čas pro výpočet součinu $\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$ změřen pro sekvenční, paralelní OpenMP i paralelní C++ variantu po sobě během jednoho běhu měřicí aplikace. Problém však je, že po dokončení paralelní oblasti v OpenMP mohou vlákna po určitou dobu zůstat ve stavu aktivního čekání a čerpat tak procesorový čas. [14] Tuto vlastnost lze sice do určité míry ovlivnit nastavením proměnné prostředí `OMP_WAIT_POLICY`, nicméně daná implementace OpenMP není povinná toto nastavení brát v potaz. Pokud by byla spuštěna paralelní C++ varianta SpGEMM výpočtu hned po dokončení OpenMP varianty, mohlo by se stát, že OpenMP vlákna budou stále aktivní a budou tak zkreslovat měření pro paralelní C++ variantu. Navíc vlákna vytvořená v rámci paralelních oblastí OpenMP většinou nezanikají po skončení paralelní oblasti, ale čekají v interní zásobárně vláken. Protože paralelní C++ varianta používá pro paralelizaci vlastní zásobárnu vláken (viz. kapitoly 8.3 a 9.2), po zahájení jejího SpGEMM výpočtu by tak aplikace pro měření používala dvojnásobek vláken nastavených pro daný SpGEMM výpočet, což by mohlo rovněž zkreslovat měření. Z výše uvedených důvodů byla proto aplikace pro měření implementována tak, že pro daný vstup je změřena doba běhu pouze pro jednu z implementovaných variant, přičemž konkrétní varianta je zvolena prostřednictvím přepínače příkazové řádky. Tím je zajištěno, že se paralelní varianty nebudou vzájemně ovlivňovat.

Aby měla naměřená data co nejvyšší vypovídací hodnotu, je pro každou vstupní matici změřen čas výpočtu pouze samotného SpGEMM násobení, tj. rozdíl časů na konci a na začátku SpGEMM výpočtu. Do měřeného času tedy není zahrnuto načítání matice ze vstupního Matrix Market souboru do struktury pro COO matice, ani její následná konverze do CSR formátu. Dále není v rámci měření použito řazení nenulových prvků v rámci řádku výstupní CSR matice, ani není použita konverze pro odstranění nulových prvků z alge-

braického vynulování. Zápis výsledné matice do Matrix Market souboru pak v rámci aplikace pro měření také není použit, neboť jde, stejně jako u načítání, o vstupně-výstupní operaci, která s výpočetní náročností samotného SpGEMM násobení nijak nesouvisí.

V OpenMP variantě jsou vlákna vytvořena automaticky až při první paralelní oblasti, zatímco vlákna v paralelní C++ variantě musí být vytvořena ještě před zahájením SpGEMM výpočtu, protože je nezbytné předat konstruktoru třídy `par_hash_spgemm` ukazatel na instanci vlastní zásobárny vláken. Vytvoření vláken v paralelní C++ variantě tedy není zahrnuto do měřeného času. Aby nebyla OpenMP varianta znevýhodněna tím, že by vytvoření jejích vláken bylo započítáno do měřeného času, je v měřicí aplikaci pro konverzi COO matice, která byla načtena ze vstupního Matrix Market souboru, do CSR formátu použita odpovídající varianta této třídy pro konverzi. Pro OpenMP variantu je tedy první paralelní oblast již součástí konverze COO matice na CSR matici, a protože není tento výpočet zahrnut do měřeného času, je vytvoření OpenMP vláken, stejně jako u paralelní C++ varianty, mimo rozsah měření. Tím je zajištěno, že nebude OpenMP varianta nijak penalizována.

Aplikace pro měření přijímá prostřednictvím povinného pozičního argumentu cestu k Matrix Market souboru matice, pro kterou má být změřen čas SpGEMM výpočtu. Pomocí volitelných argumentů lze specifikovat variantu SpGEMM násobení, která má být změřena, a také počet vláken pro paralelní varianty SpGEMM násobení. Aplikace měří reálný čas prostřednictvím hodin `std::chrono::steady_clock` a změřený čas zapisuje na standardní výstup. Více o používání této aplikace je možné nalézt v uživatelské příručce v příloze A.

Aby nebylo nutné spouštět na fakultním klastru úlohu pro měření ručně pro každou matici určenou k měření a pro každou variantu SpGEMM násobení, byl pro automatizaci měření implementován Bash skript `measurements.sh`. Tento skript přijímá jako první argument název fronty, která má být pro měřicí úlohy použita, a jako druhý argument skript přijímá cestu k adresáři s Matrix Market maticemi, které mají být použity pro měření. Skript pro každou matici z poskytnutého adresáře zařadí do specifikované fronty úloh spuštění měřicí aplikace pro každou ze tří variant SpGEMM násobení.

Jak bylo popsáno výše, aplikace `measurements` umožňuje prostřednictvím CLI argumentu specifikovat počet vláken, která se mají účastnit výpočtu. Vzhledem k tomu, že měření budou provedena na dvou typech výpočetních uzlů fakultního klastru ClusterFIT, bude počet vláken pro paralelní varianty SpGEMM násobení nastaven skriptem `measurements.sh` automaticky podle typu uzlu, pro který jsou úlohy určeny. Každý uzel s procesorem Fujitsu A64FX má k dispozici celkem 48 výpočetních jader, a proto skript pro úlohy určené pro tyto uzly automaticky nastavuje počet vláken na hodnotu 48 pro paralelní varianty výpočtu. Obdobně, měřicím úlohám určeným pro výpočetní uzel s procesorem AMD EPYC 7543P skript automaticky nastavuje paralelním variantám SpGEMM výpočtu počet vláken na hodnotu 64.

11.3 Výsledky a vyhodnocení

Kompletní výsledky měření doby výpočtu, včetně metrik paralelní ceny, zrychlení a efektivnosti, jsou uvedeny v příloze C této práce. Doby výpočtu pro matice z kategorie středních, velkých a velmi velkých matic ukazují grafy 11.1, 11.2 a 11.3.

Z výsledků pro kategorii velmi malých matic vyplývá, že použití paralelních algoritmů pro matice z této kategorie a s hustotou pod 1 % je kontraproduktivní, neboť na těchto maticích jsou paralelní varianty výpočtu pomalejší než sekvenční výpočet. Tyto výsledky však nejsou překvapivé, neboť je zřejmé, že pokud má matice takto nízký řád a pouze několik nenulových prvků v každém řádku, pak režie spojená s paralelním výpočtem výrazně převyšuje dobu samotného aritmetického výpočtu, neboť paralelní varianty kromě správy, synchronizace a plánování vláken také potřebují určit počet aritmetických operací pro každý řádek výstupní matice a na základě tohoto výsledku rozdělit výpočet řádků mezi jednotlivá vlákna.

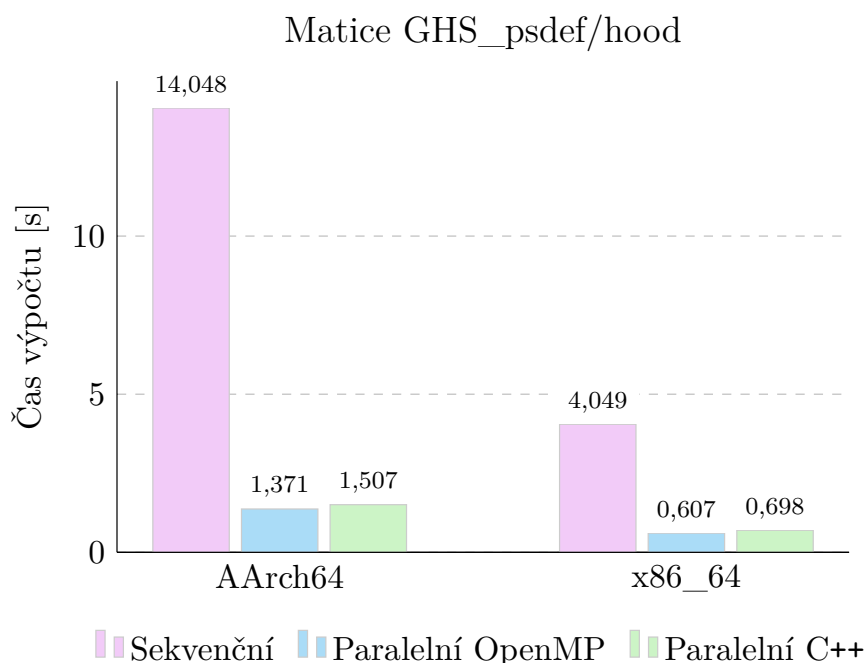
Nicméně pokud hustota stoupne nad 1 %, pak i v kategorii velmi malých matic dosahují paralelní varianty SpGEMM výpočtu nemalého zrychlení. Například pro matici Nemeth/nemeth23 s hustotou přibližně 1,7 % dosahuje OpenMP řešení přibližně dvanáctinásobného zrychlení pro oba typy uzlů, resp. téměř jedenáctinásobného zrychlení v případě paralelní C++ varianty na AArch64 uzlu a sedminásobného zrychlení na x86_64 uzlu. V případě matic s vyšší hustotou je zrychlení ještě vyšší, např. pro matici GHS_indef/exdata_1 s hustotou 6,3 % dosahují obě paralelní varianty na uzlu AArch64 více než dvacetinásobného zrychlení, na uzlu x86_64 pak téměř šestnáctinásobného zrychlení.

Takto vysoká hustota se však ve vybraných maticích vyskytuje pouze v kategorii velmi malých matic. Pro matice vybrané pro měření platí, že se stoupajícím řádem významně klesá i hustota matice. Avšak příznivým výsledkem je, že pro matice ze všech zbývajících kategorií jsou paralelní varianty SpGEMM výpočtu vždy rychlejší, a to i pro matice s velmi nízkou hustotou.

V kategorii malých matic mají všechny matice hustotu pod 0,5 %, vyjma matice Gupta/gupta3 s hustotou přes 3 %. Nejvyššího zrychlení dosahují paralelní varianty právě pro matici Gupta/gupta3, pro kterou dosáhly paralelní varianty na AArch64 uzlu více než čtrnáctinásobného zrychlení, resp. více než sedmnáctinásobného zrychlení na uzlu x86_64. Nejnižší zrychlení paralelních variant bylo v kategorii malých matic pozorováno u matice Gaertner/pesa, pro kterou obě varianty na uzlu AArch64 nedosahují ani dvojnásobného zrychlení. Na x86_64 uzlu dosahují paralelní varianty pro tuto matici také nejnižšího zrychlení, konkrétně téměř třínásobného v případě OpenMP varianty, a v případě C++ varianty bylo dokonce pozorováno paralelní zpomalení. Tento jev je opět zapříčiněn převážením paralelní režie, neboť tato matice má ve své kategorii nejnižší řád s hodnotou 11 738 a současně nejnižší hustotu, která je 0,06 %. Pro ostatní matice z kategorie malých matic dosahují paralelní ře-

šení čtyřnásobného až osminásobného zrychlení na uzlu AArch64 a přibližně trojnásobného až sedminásobného zrychlení na uzlu x86_64.

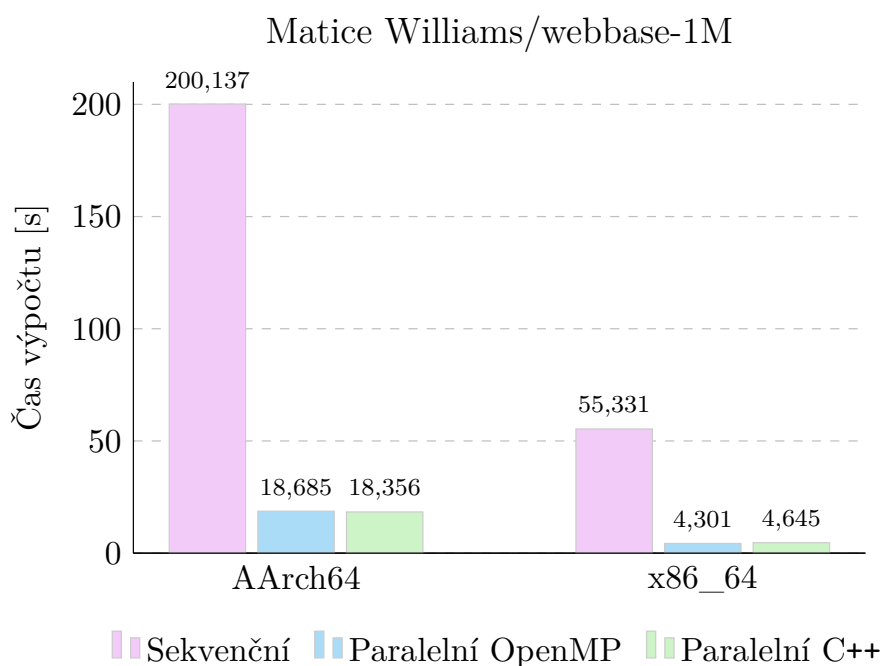
Pro matice v kategorii středních matic dosahuje hustota nejvýše hodnoty 0,05 %, avšak i pro takto nízké hustoty dosahují paralelní varianty poměrně dobrého zrychlení. Nejpomalejší byl výpočet pro matici Chevron/Chevron3, pro kterou dosáhlo OpenMP řešení více než čtyřnásobného zrychlení a paralelní C++ varianta více než trojnásobného zrychlení na AArch64 uzlu. Na x86_64 uzlu pak pro tuto matici bylo pozorováno téměř trojnásobné zrychlení v případě OpenMP varianty a téměř jedna a půlnásobné zrychlení v případě C++ varianty. Nejlepších výsledků v této kategorii bylo dosaženo u matic Hamm/scircuit a Pajek/patents_main. Na uzlu AArch64 dosahuje OpenMP varianta přibližně dvacetinásobného zrychlení, zatímco C++ varianta šestnáctinásobného, resp. patnáctinásobného zrychlení. Na uzlu x86_64 pak pro tyto matice dosahuje OpenMP řešení více než jedenáctinásobného zrychlení a C++ varianta více než sedminásobného, resp. čtyřnásobného zrychlení. Porovnání časů výpočtu pro matici z této kategorie ukazuje graf 11.1.



Obrázek 11.1 Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici GHS_psdef/hood z kategorie středních matic.

V kategorii velkých matic dosahuje hustota hodnoty $7,34 \cdot 10^{-3}$ % a nejvyššího zrychlení dosahují paralelní varianty na maticích VLSI/vas_stokes_4M a Williams/webbase-1M, pro které dosahují přibližně desetinásobného zrychlení na uzlu AArch64. Na uzlu x86_64 dosahuje OpenMP řešení pro tyto

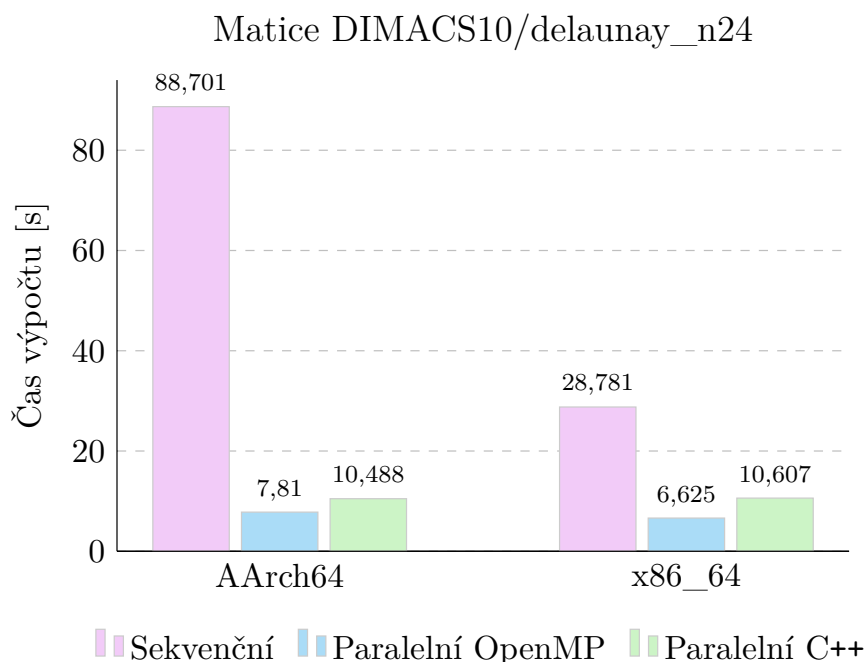
matice přibližně pětinasobného, resp. třináctinasobného zrychlení a C++ varianta téměř pětinasobného, resp. dvanáctinasobného zrychlení. U ostatních matic z této kategorie dosahují paralelní varianty přibližně čtyřnasobného až osminásobného zrychlení na AArch64 uzlu a přibližně dvojnásobného až pětinasobného zrychlení na uzlu x86_64, s výjimkou matice Bourchtein/atmosmodm, pro kterou na x86_64 uzlu nedosáhla paralelní C++ varianta téměř žádného zrychlení. Graf zachycující doby výpočtu pro matici Williams/webbase-1M ukazuje obrázek 11.2.



Obrázek 11.2 Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici Williams/webbase-1M z kategorie velkých matic.

Matice vybrané z kategorie velmi velkých matic mají obzvlášť nízkou hustotu, která nepřesahuje hodnotu $9,42 \cdot 10^{-5} \%$. V této kategorii jsou rozdíly v paralelním zrychlení nejvyšší, neboť zatímco pro matici GAP/GAP-road dosáhla OpenMP varianta na uzlu AArch64 více než pětinasobného zrychlení, pro matici FlowIPM22/Spielman_k400 dosáhla na stejném uzlu více než čtyřicetinasobného, tj. téměř lineárního zrychlení. Obdobně, paralelní C++ varianta pro tytéž matice na AArch64 uzlu dosáhla téměř trojnásobného a více než třicetipětinasobného zrychlení. Na uzlu x86_64 pro matici GAP/GAP-road dosáhla OpenMP varianta přibližně jedna a půlnásobného zrychlení, zatímco pro C++ variantu bylo dokonce pozorováno paralelní zpomalení. Pro matici FlowIPM22/Spielman_k400 bylo na uzlu x86_64 dosaženo téměř osmnáctinasobného zrychlení v případě OpenMP varianty a téměř třináctinasobného

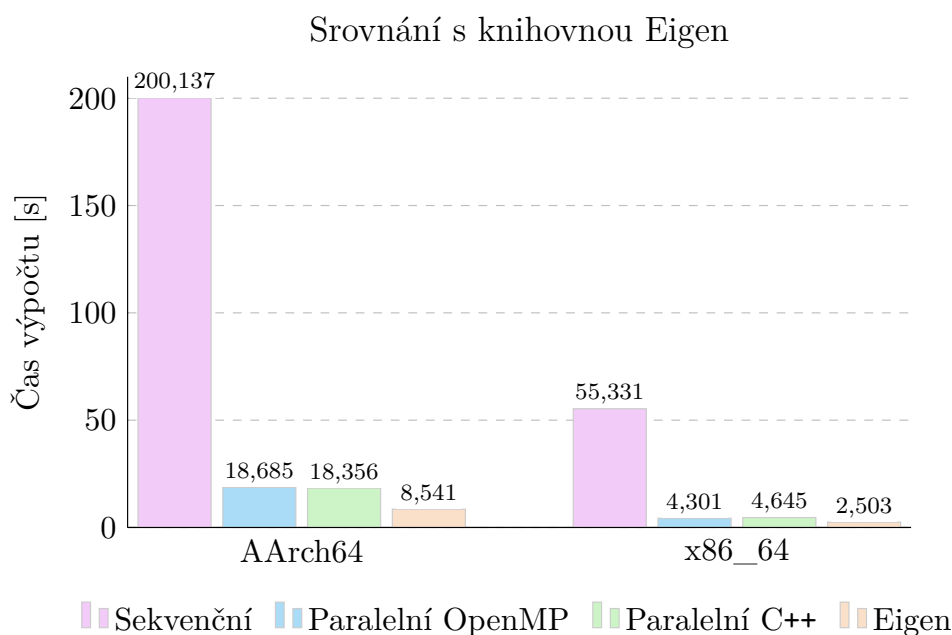
zrychlení v případě C++ varianty. Časy výpočtu pro matici DIMACS10/delaunay_n24 z kategorie velmi velkých matic je možné vidět na grafu 11.3.



Obrázek 11.3 Časy výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 pro matici DIMACS10/delaunay_n24 z kategorie velmi velkých matic.

Pro objektivní zhodnocení implementovaných variant SpGEMM výpočtu bylo provedeno srovnání s vysoce optimalizovanou C++ knihovnou Eigen [83]. Měření bylo provedeno pro matici Williams/webbase-1M na obou architekturách, přičemž výsledky tohoto srovnání jsou k vidění v grafu 11.4. Z uvedených hodnot je zřejmé, že knihovna Eigen dosahuje významně lepších výsledků. Na uzlu AArch64 dokončila knihovna Eigen výpočet za 8,541 vteřin a na uzlu x86_64 dokonce za 2,503 vteřin. Těmito hodnotami knihovna nejenom překonala sekvenční variantu SpGEMM výpočtu, ale dokonce překonala i obě paralelní řešení využívající desítky výpočetních vláken. Ačkoliv byla knihovna Eigen kompilována s podporou OpenMP, podle dokumentace knihovny není násobení řídkých matic prováděno paralelně. Tento výrazný rozdíl v časech výpočtu je dán nejenom vysokou optimalizací samotného SpGEMM násobení v knihovně Eigen, ale také využitím vektorizace výpočtu. Pro akumulaci nenulových prvků využívá knihovna Eigen řídký akumulátor (viz. kapitola 7.3.1), který disponuje konstantním časem přístupu k akumulovaným prvkům. Vzhledem k tomu, že knihovna Eigen podle dokumentace používá pro SpGEMM výpočet pouze jedno vlákno, dokáže značnou část tohoto akumulátoru udržet v cache paměti procesoru, a to i pro velké matice. Eigen dále využívá vektorové

instrukce procesoru na rozdíl od implementovaných řešení, které spoléhají na standardní iterativní zpracování. Díky vektorizaci výpočtu je pak možné vykonat několik iterací smyčky daného algoritmu najednou, což zcela zásadně zvyšuje výkon celého výpočtu.



Obrázek 11.4 Srovnání časů výpočtu všech variant SpGEMM násobení na výpočetních uzlech typu AArch64 a x86_64 s knihovnou Eigen pro matici Williams/webbase-1M z kategorie velkých matic.

Paralelní efektivnost se pro většinu matic pohybuje mezi 10 % a 20 % na AArch64 uzlu, na uzlu x86_64 se hodnoty většinou pohybují pod 10 %. Avšak tento výsledek je důsledkem velmi vysokého počtu použitých jader, neboť paralelní efektivnost je dána podílem sekvenčního času a součinu počtu jader s paralelním časem, přičemž na uzlech s procesorem Fujitsu A64FX bylo použito všech 48 fyzických jader a na uzlu s procesorem AMD EPYC 7543P všech 64 logických jader. Pokud byl pro některé matice počet použitých vláken experimentálně snížen na 32 nebo dokonce 16 vláken, tak došlo sice k přibližně 30% a v některých případech více než 40% zpomalení oproti využití všech dostupných jader, avšak efektivita díky snížení počtu vláken stoupla na 50 % a v některých případech dokonce téměř na 60 %. Z těchto výsledků tedy plyne, že problém násobení řídkých matic je velmi špatně škálovatelný, neboť vyšší počet výpočetních jader sice vede k rychlejšímu výpočtu, avšak paralelní efektivnost při zvýšení počtu použitých jader naopak značně klesá.

Tato špatná škálovatelnost je následkem vlastností samotného problému násobení řídkých matic. Výpočetní problémy lze z hlediska efektivní paralelizovatelnosti na architekturách se sdílenou pamětí rozdělit na výpočetně inten-

zivní a paměťově intenzivní problémy. Pro výpočetně intenzivní problémy je typické, že vykazují vysokou tzv. aritmetickou hustotu, která vyjadřuje poměr celkového počtu aritmetických operací k celkové velikosti přenesených dat. Vysoká aritmetická hustota tedy znamená, že čas procesoru strávený výpočtem nad daty je výrazně vyšší než čas potřebný k přesunu těchto dat z paměti do procesoru. Naopak pro paměťově intenzivní problémy je typická nízká aritmetická hustota, tj. čas procesoru strávený výpočtem nad danými daty je výrazně nižší než čas potřebný k přesunu těchto dat do CPU. U paměťově intenzivních problémů proto výpočetní jádra většinu procesorového času tráví čekáním na data, než samotným výpočtem. [3, 84]

Násobení řídkých matic patří mezi paměťově intenzivní problémy, navíc s nepravidelným způsobem přístupu do paměti. [85] Výkon SpGEMM algoritmu je tedy zásadně omezen rychlostí, s jakou lze přenášet data z hlavní paměti do procesoru. Paměťová propast, která byla popsána v kapitole 2.2, je proto kvůli nepravidelnému způsobu přístupu do paměti, způsobenému nepřímým adresováním, obzvláště omezující, neboť ačkoliv je k prvkům v řádku matice **A** přistupováno souvisle, sloupcové indexy těchto prvků určují, které řádky matice **B** je potřeba načíst z hlavní paměti. Toto vede nejen k nesouvislým přístupům do paměti, ale také k velmi omezené úspěšnosti speciálních jednotek procesoru, které předpovídají budoucí přístupy do paměti, neboť tyto přístupy jsou z hlediska hardwaru nepředvídatelné. Z těchto důvodů může nižší paralelní zrychlení v naměřených datech reflektovat situaci, kdy souběžná vlákna maximálně využila dostupnou paměťovou propustnost. Tuto hypotézu podporuje i porovnání naměřených výsledků na AArch64 a x86_64 architekturách.

Výpočetní uzel s procesorem AMD EPYC 7543P dosahuje ve většině případů výrazně kratších výpočetních časů než uzel s procesorem Fujitsu A64FX, a to jak v sekvenční variantě, tak v paralelních variantách SpGEMM výpočtu. Tento rozdíl je dán především odlišnými hardwarovými specifikacemi, neboť procesor AMD EPYC 7543P disponuje výrazně vyšší taktovací frekvencí, která dosahuje až 3,7 GHz v porovnání s 2,2 GHz procesoru Fujitsu A64FX, a větší velikostí cache paměti, která je pro nejvyšší úroveň procesoru AMD EPYC 7543P rovna 256 MB a pro Fujitsu A64FX 32 MB. Sekvenční výpočet na x86_64 uzlu proto efektivně zužitkovává velmi výkonná výpočetní jádra a větší kapacitu cache paměti. [9, 81]

Z naměřených výsledků je však zřejmé, že ačkoliv má uzel s procesorem AMD EPYC 7543P vyšší počet procesorových jader a dosahuje výrazně nižších výpočetních časů, jsou hodnoty paralelního zrychlení na tomto uzlu značně nižší než na uzlu s procesorem Fujitsu A64FX. Tento jev názorně ilustruje paměťovou intenzitu SpGEMM problému a úzké hrdlo paměťové propasti, neboť zatímco AMD EPYC 7543P má paměťovou propustnost 204,8 GB/s, procesor Fujitsu A64FX podle technických specifikací disponuje paměťovou propustností 1024 GB/s. [9, 81] Uzel s procesorem AMD EPYC 7543P tedy sice dosahuje nižších časů sekvenčního výpočtu, ale při použití všech dostup-

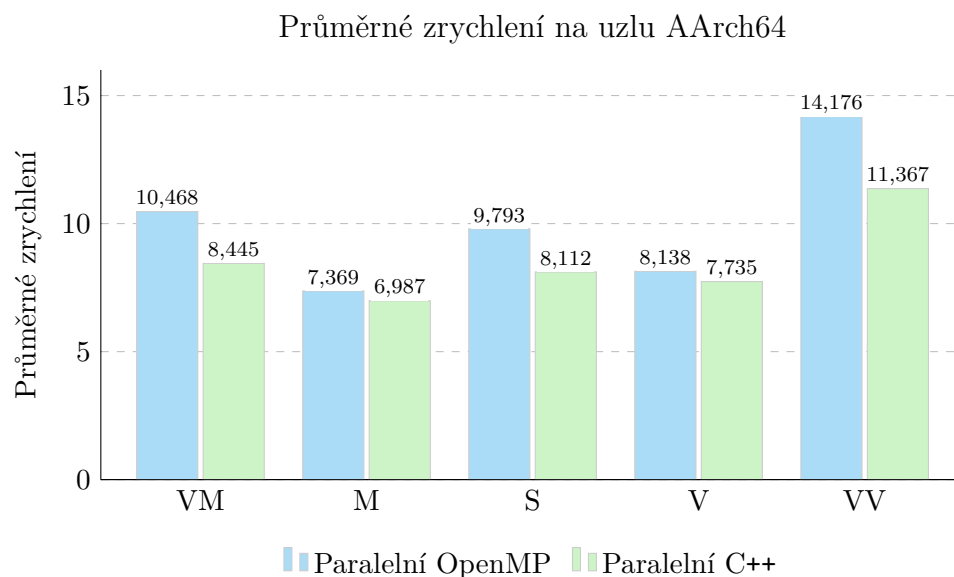
ných jader u paralelních variant dojde k zahlcení paměťové sběrnice, která kvůli nižší propustnosti nedokáže zásobovat výpočetní jádra potřebnými daty dostatečně rychle. Naproti tomu uzel s procesorem Fujitsu A64FX dosahuje horších časů sekvenčního výpočtu, neboť disponuje méně výkonnými jádry, ale díky velmi vysoké paměťové propustnosti dokáže rychleji dodávat data výpočetním jádrům. Z těchto důvodů je paralelní zrychlení na uzlu AArch64 výrazně vyšší v porovnání s uzlem x86_64. Kvůli nižší paměťové propustnosti proto SpGEMM násobení na uzlu s procesorem AMD EPYC 7543P škáluje hůře než na uzlu s procesorem Fujitsu A64FX. Tento jev zároveň potvrzuje paměťovou intenzitu SpGEMM problému, která je společně s paměťovou propastí hlavním důvodem, kvůli kterému nebylo možné dosáhnout na zvolených výpočetních uzlech lineárního zrychlení.

Tyto skutečnosti odrážejí i hodnoty průměrného zrychlení pro každou kategorii řídkých matic, které je možné nalézt v tabulkách C.16 a C.34. Hodnoty z těchto tabulek názorně zachycují grafy 11.5 a 11.6. Z výsledků průměrného zrychlení uzlu x86_64 je zřejmé, že nejvyššího průměrného zrychlení je dosaženo v kategorii velmi malých matic, konkrétně 9,2 pro OpenMP a 6,3 pro C++ variantu. S rostoucí velikostí matic však průměrné zrychlení klesá a zůstává v rozmezí přibližně 5 až 6 pro OpenMP a 4 až 5 v případě varianty s C++ vlákny. Tento propad a následné ustálení zrychlení u větších matic názorně demonstrovuje zahlcení paměťové sběrnice, která kvůli nižší paměťové propustnosti tohoto uzlu nedokáže dodávat potřebná data výpočetním jádrům dostatečně rychle. Na uzlu AArch64 dosahují průměrná zrychlení výrazně vyšších hodnot v porovnání s uzlem x86_64 a to obzvlášť v kategoriích středních, velkých a velmi velkých matic. Tento jev proto názorně ilustruje výhodu velmi vysoké paměťové propustnosti, kterou uzel AArch64 disponuje.

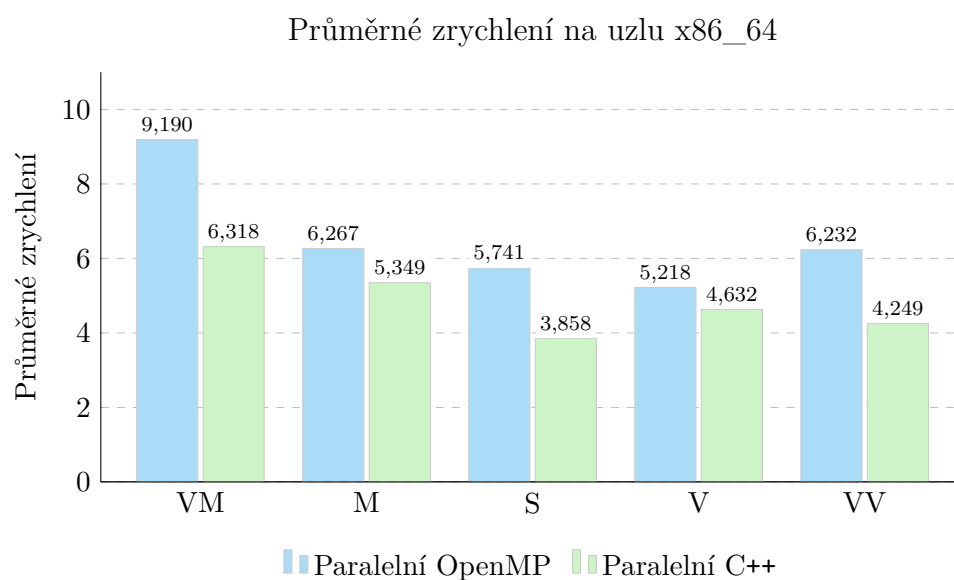
Pro detailnější analýzu nižšího průměrného zrychlení v závislosti na hardwarových specifikacích bylo dále provedeno měření časů výpočtu a dosaženého zrychlení při použití různých počtů vláken. Tato doplňující měření byla provedena pro matici Williams/webbase-1M ze skupiny velkých matic na obou typech výpočetních uzlů pro 4, 8, 16, 32 a 48 vláken, které odpovídají maximálnímu počtu fyzických jader uzlu s procesorem Fujitsu A64FX. Naměřené hodnoty jsou uvedeny v tabulkách C.17 a C.18 pro uzel AArch64 a v tabulkách C.35 a C.36 pro uzel x86_64. Pro lepší přehlednost jsou závislosti zrychlení na počtu použitých vláken vizualizovány také v grafech 11.7 a 11.8.

Při použití 4 vláken lze na obou uzlech pozorovat paralelní zpomalení. Na uzlu AArch64 bylo dosaženo zrychlení přibližně 0,7 jak pro OpenMP, tak pro C++ variantu SpGEMM výpočtu. Obdobné výsledky lze pozorovat i pro uzel x86_64, kde je zrychlení při použití 4 vláken přibližně 0,76 pro OpenMP a 0,79 pro variantu s C++ vlákny. Z těchto výsledků je zřejmé, že při takto nízkém počtu použitých vláken jsou přínosy paralelizace převáženy paralelní režií, kvůli které je paralelní výpočet pomalejší než v sekvenční variantě.

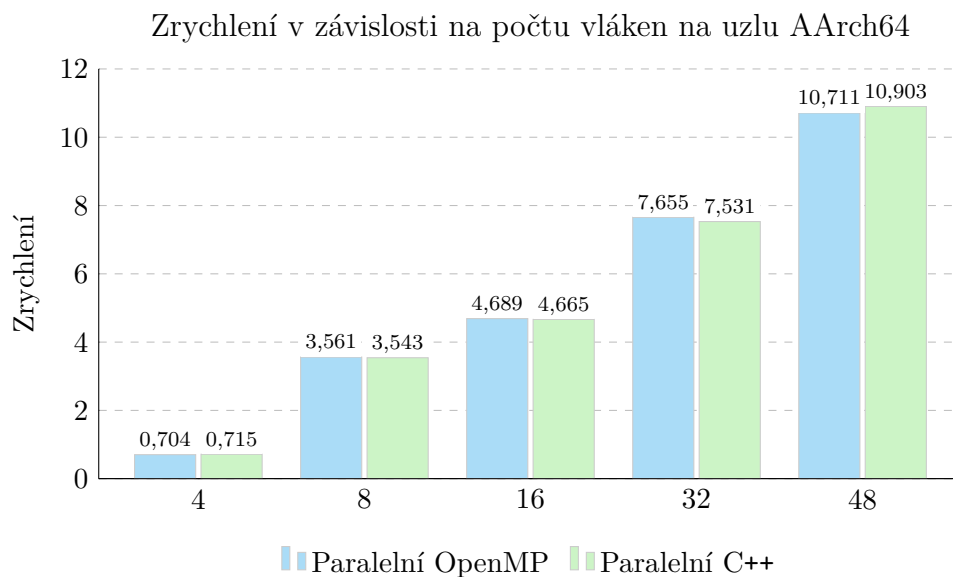
Přínosy paralelizace se na obou architekturách začínají projevovat při použití 8 vláken, pro která oba typy uzlů dosáhly zrychlení s hodnotou přibližně



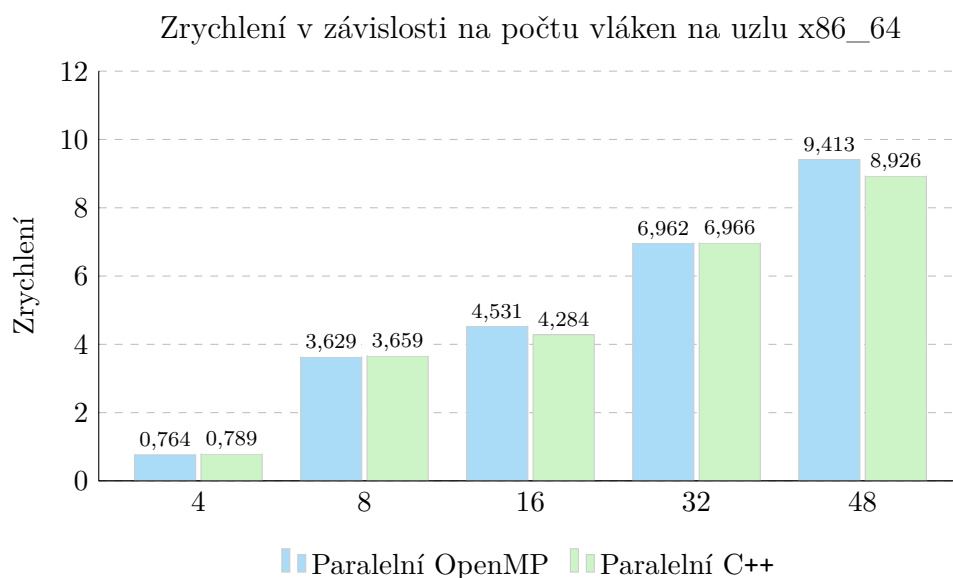
Obrázek 11.5 Průměrné zrychlení paralelních variant SpGEMM násobení pro jednotlivé skupiny řídkých matic na výpočetním uzlu s procesorem Fujitsu A64FX. Označení na horizontální ose odpovídají zkratkám skupin řídkých matic.



Obrázek 11.6 Průměrné zrychlení paralelních variant SpGEMM násobení pro jednotlivé skupiny řídkých matic na výpočetním uzlu s procesorem AMD EPYC 7543P. Označení na horizontální ose odpovídají zkratkám skupin řídkých matic.



■ **Obrázek 11.7** Zrychlení paralelních variant SpGEMM násobení v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu s procesorem Fujitsu A64FX.



■ **Obrázek 11.8** Zrychlení paralelních variant SpGEMM násobení v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu s procesorem AMD EPYC 7543P.

3,5. Z naměřených hodnot však plyne, že při dalším zvyšování počtů vláken není nárůst výkonu lineární a se zvyšujícím se počtem vláken nárůst výkonu dokonce zpomaluje. Rozdíl ve škálovatelnosti mezi architekturami je nejvíce zřejmý právě při vyšším počtu použitých vláken. Zatímco na uzlu AArch64 dosahuje OpenMP varianta při využití 48 vláken zrychlení 10,711, na uzlu x86_64 je při využití 48 logických jader toto zrychlení znatelně nižší s hodnotou 9,413. Tyto výsledky opět korespondují s pamětovou intenzitou SpGEMM problému, pamětovou propastí a omezující prostupností pamětové sběrnice. Na uzlu s procesorem AMD EPYC 7543P a propustností pamětové sběrnice 204,8 GB/s proto nastane zahlcení pamětové sběrnice dříve a s rostoucím počtem vláken se doba čekání na data propisuje do zpomalení růstu celkového zrychlení výrazněji než na uzlu AArch64, který disponuje pamětovou propustností 1024 GB/s.

Pro většinu matic platí, že OpenMP řešení je rychlejší než varianta s C++ vlákny, která dokázala překonat OpenMP řešení jen v hrstce specifických případů. Tento rozdíl ve výkonu paralelních variant je nejvýraznější v kategorii velmi velkých matic, avšak příznivým výsledkem je, že pro zbývající kategorie je tento rozdíl většinou poměrně malý, což je při zohlednění faktu, že OpenMP je vysoce optimalizovaná knihovna, která je ve vývoji téměř 30 let, přijatelný výsledek. Úzkým hrdlem paralelní C++ varianty je zřejmě způsob synchronizace použitý v zásobárně vláken. V implementovaném řešení je pro globální i privátní fronty úloh použitý jeden společný zámek. Ve třídě `par_hash_spgemm`, která zajišťuje samotné násobení řídkých matic, jsou úlohy jednotlivým vláknům rozděleny prostřednictvím metody zásobárny vláken `for_each_thread`, která je ekvivalentem paralelní oblasti z OpenMP. Tato metoda vloží do privátní fronty každého vlákna úlohu, která byla této metodě předána prostřednictvím jejího argumentu. Tím, že je souběžný přístup do všech privátních front chráněn stejným zámkem, je vyzvednutí úloh z těchto front ve výsledku částečně serializováno, ačkoliv samotný výpočet těchto úloh již probíhá paralelně. Toto zpomalení při vkládání a vyzvedávání úloh z privátních front jednotlivých vláken je proto příčinou horšího výkonu paralelní C++ varianty v porovnání s OpenMP variantou.

Na závěr vyhodnocení je dále vhodné poznamenat, že ačkoliv jsou paralelní algoritmy standardní knihovny jazyka C++ kvůli omezením popsaným v kapitolách 3.1.6 a 8.3 nevhodné pro vědecká měření, nelze z toho usuzovat, že by v porovnání s knihovnou OpenMP selhaly. Z návrhu paralelních algoritmů STL totiž vyplývá, že tyto algoritmy nejsou zamýšleny jako konkurence specializovaným knihovnám pro paralelizaci, jako je OpenMP, ani jako jejich náhrada. Jejich cílem je poskytnout možnost snadné paralelizace vybraných částí zdrojového kódu, přičemž díky důmyslnému návrhu je dopad přepracování tohoto zdrojového kódu minimální, neboť pro přechod na paralelní variantu daného algoritmu stačí předat jako první argument požadovanou strategii provádění. Paralelní algoritmy STL a knihovna OpenMP tedy míří na odlišné cílové skupiny uživatelů.

V rámci této práce bylo empiricky ověřeno, že určité konstrukty knihovny OpenMP lze plně nahradit prostředky jazyka C++ s využitím vlastní implementace zásobárny vláken. Tento přístup má však dva zásadní nedostatky. Prvním je značná implementační náročnost samotného návrhu zásobárny vláken jakožto centrálního prvku odpovědného za distribuci výpočtu mezi jednotlivá vlákna. Druhým je výkonnostní omezení vlastního řešení, neboť naměřená data ukazují, že se výsledná implementace dokázala výkonu OpenMP pouze přiblížit a překonat jej jen v několika ojedinělých případech. Vzhledem k tomu, že optimalizace takové implementace představuje časově mimořádně náročný proces, je pro paralelizaci výpočetně náročných úloh vhodnější použít prověřené specializované řešení. Výsledky této práce tak potvrzují, že knihovny typu OpenMP zůstávají pro vědecké výpočty a vysoce specializovanou paralelizaci nejvhodnější volbou.

Závěr

Hlavním cílem této diplomové práce byla komplexní analýza paralelních algoritmů pro efektivní násobení řídkých matic na výpočetních architekturách se sdílenou pamětí a následný návrh, implementace, testování, měření a vyhodnocení algoritmu pro násobení řídkých matic v sekvenční variantě, v paralelní variantě využívající OpenMP a v paralelní variantě využívající pouze standardní knihovnu jazyka C++.

V teoretické části práce byly podrobně prozkoumány principy fungování vícejádrových procesorů, byla popsána paměťová hierarchie, paměťová koherence a problém falešného sdílení. Následně byly představeny softwarové možnosti pro paralelizaci nad sdílenou pamětí, konkrétně prostředky standardní knihovny jazyka C++ včetně moderních prvků ze standardu C++20, a také programovací model a aplikační rozhraní knihovny OpenMP. Další část teoretické rešerše byla věnována detailní analýze všech formulací násobení matic, včetně vhodnosti těchto formulací pro paralelní násobení řídkých matic. Významnou část rešerše dále tvoří komplexní analýza formátů řídkých matic pro jejich efektivní reprezentaci v paměti a také souborových formátů pro jejich ukládání a přenos. Dále byla zpracována zevrubná analýza postupů algoritmů násobení řídkých matic, která zmapovala různé metody predikce velikosti výsledné matice, strategie rozdělení výpočetní zátěže napříč vlákny a různé typy akumulátorů pro efektivní ukládání mezivýsledků.

Praktická část práce na teoretickou řešerši navázala návrhem a implementací paralelního algoritmu, který rozděluje výpočet do několika na sebe navazujících logických fází. Pro účely paralelizace výpočtu využívajícího pouze standardní jazyk C++ byla implementována vlastní zásobárna vláken a dále byla navržena a implementována specializovaná hešovací tabulka, která byla použita pro implementaci akumulátoru s hešováním. Cíle práce bylo dosaženo implementací celkem tří variant SpGEMM výpočtu. První variantou je sekvencí implementace, druhou variantou je paralelní verze využívající knihovnu OpenMP a třetí variantou je paralelní verze využívající implementovanou zásobárnu vláken pro paralelizaci výpočtu. Při vývoji byly zohledněny také tech-

niky pro eliminaci falešného sdílení a byly vysvětleny optimalizace překladače pro předávání vypočítaných výsledků prostřednictvím návratových hodnot. Korektnost vytvořených CLI aplikací byla zajištěna implementací robustních testů v testovacím frameworku Doctest. Veškeré procesy spojené s kompilací, spouštěním testů a generováním dokumentace byly plně automatizovány prostřednictvím nástroje CMake. Tyto procesy byly dále integrovány do pipeline kontinuální integrace na fakultní instanci systému GitLab, včetně automatizace nasazení vygenerované dokumentace na platformu PagesFIT.

V závěrečné fázi proběhlo zevrubné výkonnostní měření na výpočetních uzlech fakultního klastru ClusterFIT. Měření bylo provedeno na výpočetních uzlech s procesorem Fujitsu A64FX a také na uzlu s procesorem AMD EPYC 7543P. Pro účely měření bylo stanoveno pět kategorií řídkých čtvercových matic podle jejich řádu, a pro každou takovou kategorii byla z kolekce SuiteSparse Matrix Collection vybrána minimálně jedna matice s celočíselnými, reálnými a komplexními nenulovými prvky, pokud takové matice byly pro danou kategorii v SuiteSparse Matrix Collection k dispozici. Pro každou vybranou matici byla změřena doba výpočtu $\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$, kde $\mathbf{A} \in \mathbb{C}^{n,n}$ je vstupní čtvercová matice, a to pro sekvenční variantu, pro paralelní variantu využívající knihovnu OpenMP a pro paralelní variantu využívající pouze prostředky standardní knihovny jazyka C++. Takto naměřená data sloužila k výpočtu dalších klíčových metrik, konkrétně paralelní ceny, paralelního zrychlení a paralelní efektivnosti. Ze získaných výsledků vyplynulo, že nejlepších výsledků dosahuje varianta využívající knihovnu OpenMP. V rámci vyhodnocení byly identifikovány výhody a omezení obou implementovaných paralelizačních přístupů, přičemž byl stanoven závěr, že pro vědecká měření a úlohy vyžadující vysoce specializovanou paralelizaci je nejvhodnější volbou použití existujícího řešení typu OpenMP.

Všechny stanovené cíle této práce byly splněny. Výsledkem jsou tři varianty aplikace pro násobení řídkých matic, které umožňují objektivní porovnání vysokoúrovňového přístupu pomocí knihovny OpenMP s nízkoúrovňovou paralelizací prostřednictvím vlastní zásobárny vláken v moderním standardu jazyka C++. Vyhodnocení metrik paralelní ceny, zrychlení a efektivnosti ukázalo konkrétní výhody a omezení jak přístupu založeného na direktivách OpenMP, tak explicitní správy vláken pomocí jazyka C++. Tato diplomová práce tak poskytuje ucelenou a prakticky podloženou analýzu, která usnadní rozhodování při budoucím návrhu paralelních algoritmů pro operace nad řídkými maticemi.

12.1 Možnosti rozšíření

Přestože byly splněny všechny vytyčené cíle, je oblast paralelizace násobení řídkých matic natolik rozsáhlá, že nabízí hned několik možností dalšího rozšíření této diplomové práce.

První velmi zajímavou možností rozšíření této práce je rozšíření paralelních variant násobení řídkých matic na distribuované verze prostřednictvím

standardu pro posílání zpráv MPI. Hybridní přístup, který by kombinoval paralelizaci uvnitř jednoho výpočetního uzlu pomocí OpenMP nebo C++ vláken se síťovou komunikací napříč jednotlivými uzly klastru, by umožnil distribuované násobení matic velmi velkých rozměrů s vysokým počtem nenulových prvků, kvůli kterým by se tyto řídké matice nevešly do operační paměti jednoho výpočetního uzlu. Pro tuto distribuovanou variantu by musela být věnována zvláštní pozornost grafovým metodám rozdělení výpočetní zátěže, aby byla udržena komunikace mezi uzly výpočetního klastru na co nejnížší úrovni.

Druhou možností rozšíření je implementace dalších typů akumulátorů pro akumulaci mezivýsledků, jako je například SPA, ESC nebo akumulátory založené na haldě či slévání. Protože každá z těchto metod má odlišné nároky na paměť a výpočetní čas v závislosti na množství zpracovávaných dat, nabízí se také implementace mechanismu pro jejich dynamický výběr. Tento přístup by během výpočtu analyzoval stanovenou charakteristiku, například poměr predikce celkového počtu nenulových prvků v daném řádku výstupní matice k její velikosti, a na základě toho by byl zvolen optimální akumulátor.

Další zajímavou možností rozšíření je paralelní detekce symetrie výstupní řídké matice $\mathbf{C}$. Souborový formát Matrix Market podporuje uložení matic obecných, symetrických, antisymetrických a Hermitovských. Pro efektivní uložení matic dokáže formát Matrix Market využít vlastnosti plynoucí z typu symetrie takové matice, například pro symetrickou matici jsou vynechány všechny prvky pod nebo nad hlavní diagonálou. Toto rozšíření by pro matici v některém z řídkých formátů, jako je např. CSR, CSC nebo COO, detekovalo druh její symetrie, přičemž samotný výpočet by byl paralelizován s využitím knihovny OpenMP nebo zásobárny vláken implementované v rámci této diplomové práce.

Uživatelská příručka

Tato diplomová práce poskytuje tři varianty aplikace pro násobení řídkých matic. První varianta je sekvenční verze `seq_spgemm`, druhá varianta je paralelní verze `omp_spgemm` využívající OpenMP a třetí varianta `par_spgemm` využívá pro paralelizaci C++ vlákna. Pro všechny tři varianty platí, že obě vstupní matice musí být ve formátu Matrix Market.

Součástí projektu je také samostatná aplikace pro měření `measurements`. Tato aplikace měří čas výpočtu $\mathbf{A}^2 = \mathbf{A} \cdot \mathbf{A}$, kde $\mathbf{A} \in \mathbb{C}^{n,n}$ je vstupní čtvercová matice. Konkrétní variantu SpGEMM výpočtu, která se má pro měření použít, lze specifikovat daným argumentem z příkazové řádky.

A.1 Závislosti

A.1.1 Povinné závislosti

- C++ překladač
 - Kompilátor musí podporovat standard C++20 a mít základní podporu OpenMP 5.0
 - Například:
 - * GCC 13 a novější
 - * Clang 17 a novější
 - * MSVC 19.29 a novější
- CMake
 - Vyžadována verze 3.22.1 a novější
 - <https://cmake.org/>
 - Na distribucích založených na Debianu lze použít příkaz: `sudo apt-get install cmake`

- Předkompilované binární soubory jsou ke stažení na oficiálních stránkách na odkazu: <https://cmake.org/download/>

A.1.2 Volitelné závislosti

Následující závislosti jsou volitelné, což znamená, že všechny aplikace, včetně aplikace pro měření, se bez těchto závislostí korektně zkompilují. Pokud však nebude na cílovém systému Doxygen k dispozici, nebude možné vygenerovat dokumentaci projektu, a pokud nebude k dispozici Graphviz, nebude možné vygenerovat vizualizace tříd.

- Doxygen
 - <https://www.doxygen.nl/>
 - Na distribucích založených na Debianu lze použít příkaz: `sudo apt-get install doxygen`
- Graphviz
 - <https://graphviz.org/>
 - Na distribucích založených na Debianu lze použít příkaz: `sudo apt-get install graphviz`

A.1.3 Použité externí knihovny

Všechny knihovny uvedené níže není potřeba instalovat na cílový systém. Tyto knihovny jsou staženy automaticky prostřednictvím funkce `FetchContent` nástroje CMake. Během kompilace je proto nutné připojení k internetu.

- `argparse`
 - <https://github.com/p-ranav/argparse>
 - Použito pro zpracování argumentů z příkazové řádky
- `fast_matrix_market`
 - https://github.com/alugowski/fast_matrix_market
 - Použito pro načítání Matrix Market souboru do vnitřní datové struktury
- `Doctest`
 - <https://github.com/doctest/doctest>
 - Použito pro testování implementovaných tříd

A.2 Sestavení

Projekt používá nástroj CMake, proto lze k vygenerování souborů pro sestavení použít následující příkazy:

```
mkdir build
cd build
cmake ..
```

Jakmile jsou soubory pro sestavení vygenerovány, lze projekt zkompileovat pomocí následujícího příkazu v adresáři `build`:

```
make -j $(nproc)
```

Pokud jsou na cílovém systému nainstalovány závislosti Doxygen a Graphviz, lze dokumentaci projektu vygenerovat pomocí následujícího příkazu v adresáři `build`:

```
make docs
```

V rámci kontinuální integrace je na fakultní instanci systému GitLab dokumentace projektu generována automaticky. Tato vygenerovaná dokumentace je také automaticky nasazena na PagesFIT na následujícím veřejně dostupném odkaze:

■ <https://javormat.pages.fit/ni-dip-spgemm/>

A.3 Použití

A.3.1 seq_spgemm

```
./seq_spgemm [--help] [--version] [--sort] [--prune]
              [--format FORMAT] [--output PATH] matrix-A matrix-B
```

Vypočítá součin řádkých matic `A` a `B`.

Poziční argumenty:

<code>matrix-A</code>	Cesta k souboru s maticí <code>A</code> .
<code>matrix-B</code>	Cesta k souboru s maticí <code>B</code> .

Volitelné argumenty:

<code>-h, --help</code>	Vypíše nápovědu a ukončí program.
<code>-v, --version</code>	Vypíše verzi aplikace a ukončí program.
<code>-s, --sort</code>	Seřadí nenulové prvky v řádku výstupní matice <code>C</code> podle sloupcového indexu.

<code>-p, --prune</code>	Odstraní nulové prvky vzniklé algebraickým vynulováním z řádků výstupní matice C.
<code>-f, --format FORMAT</code>	Specifikuje formát výstupní matice C. Podporované formáty: "mm" (Matrix Market), "raw" (neformátovaná data matice) [výchozí: "mm"]
<code>-o, --output PATH</code>	Specifikuje soubor pro zápis výstupní matice C. Pokud není zadán, použije se standardní výstup.

A.3.2 `omp_spgemm`

Výchozí hodnota počtu vláken je určena podle počtu dostupných jader systému, na kterém byla aplikace zkompilována.

```
./omp_spgemm [--help] [--version] [--sort] [--prune]
              [--format FORMAT] [--output PATH]
              [--threads NUM_THREADS] matrix-A matrix-B
```

Vypočítá součin řídkých matic A a B.

Poziční argumenty:

<code>matrix-A</code>	Cesta k souboru s maticí A.
<code>matrix-B</code>	Cesta k souboru s maticí B.

Volitelné argumenty:

<code>-h, --help</code>	Vypíše nápovědu a ukončí program.
<code>-v, --version</code>	Vypíše verzi a ukončí program.
<code>-s, --sort</code>	Seřadí nenulové prvky v každém řádku výstupní matice C podle sloupcového indexu.
<code>-p, --prune</code>	Odstraní nulové prvky vzniklé algebraickým vynulováním z řádků výstupní matice C.
<code>-f, --format FORMAT</code>	Specifikuje formát výstupní matice C. Podporované formáty: "mm" (Matrix Market), "raw" (neformátovaná data matice). [výchozí: "mm"]
<code>-o, --output PATH</code>	Specifikuje soubor pro zápis výstupní matice C. Pokud není zadán, použije se standardní výstup.

`-t, --threads NUM_THREADS` Specifikuje počet vláken pro paralelní výpočet. [výchozí: 48]

A.3.3 `par_spgemm`

Výchozí hodnota počtu vláken je určena podle počtu dostupných jader systému, na kterém byla aplikace zkompileována.

```
./par_spgemm [--help] [--version] [--sort] [--prune]
              [--format FORMAT] [--output PATH]
              [--threads NUM_THREADS] matrix-A matrix-B
```

Vypočítá součin řídkových matic A a B.

Poziční argumenty:

<code>matrix-A</code>	Cesta k souboru s maticí A.
<code>matrix-B</code>	Cesta k souboru s maticí B.

Volitelné argumenty:

<code>-h, --help</code>	Vypíše nápovědu a ukončí program.
<code>-v, --version</code>	Vypíše verzi aplikace a ukončí program.
<code>-s, --sort</code>	Seřadí nenulové prvky v každém řádku výstupní matice C podle sloupcového indexu.
<code>-p, --prune</code>	Odstraní nulové prvky vzniklé algebraickým vynulováním z řádků výstupní matice C.
<code>-f, --format FORMAT</code>	Specifikuje formát výstupní matice C. Podporované formáty: "mm" (Matrix Market), "raw" (neformátovaná data matice). [výchozí: "mm"]
<code>-o, --output PATH</code>	Specifikuje soubor pro zápis výstupní matice C. Pokud není zadán, použije se standardní výstup.
<code>-t, --threads NUM_THREADS</code>	Specifikuje počet vláken pro paralelní výpočet. [výchozí: 48]

A.3.4 `measurements`

Výchozí hodnota počtu vláken je určena podle počtu dostupných jader systému, na kterém byla aplikace zkompileována.

```
./measurements [--help] [--version] [--algo ALGORITHM]
               [--threads NUM_THREADS] matrix-A
```

Změří dobu výpočtu druhé mocniny řídské matice A.

Poziční argumenty:

matrix-A	Cesta k souboru s maticí A.
----------	-----------------------------

Volitelné argumenty:

-h, --help	Vypíše nápovědu a ukončí program.
-v, --version	Vypíše verzi aplikace a ukončí
-a, --algo ALGORITHM	Specifikuje variantu SpGEMM výpočtu. Podporované varianty: "seq" (sekvenční), "omp" (paralelní s OpenMP), "par" (paralelní s C++ vlákny) [výchozí: "seq"]
-t, --threads NUM_THREADS	Specifikuje počet vláken pro paralelní výpočet. Pro sekvenční variantu je tato hodnota ignorována. [výchozí: 48]

A.3.5 Automatizace měření

Pro účely automatizace měření na fakultním klastru ClusterFIT byl implementován Bash skript `measurements.sh`. Skript očekává, že binární soubor pro měření na uzlu ARM64 se nachází ve složce `build_arm` a binární soubor pro měření na uzlu AMD64 ve složce `build_amd`. Jeho použití je následující:

```
./measurements.sh queue folder [num_threads]
```

Poziční argumenty:

queue	Určuje frontu úloh, která se má použít pro měření.
folder	Adresář s maticemi, pro které se má provést měření.

Volitelné argumenty:

num_threads	Určuje počet vláken pro paralelní varianty SpGEMM výpočtu. Pokud není zadán, použije se počet vláken rovnající se počtu jader dostupných na zadaném typu uzlu.
-------------	--

A.3.6 Testy

Všechny testy lze spustit ze složky se zkompilevanými binárními soubory následujícím příkazem:

```
ctest . --output-on-failure -j $(nproc)
```

Data pro měření

Vybrané matice byly rozděleny do pěti kategorií podle svého řádu. Konkrétní meze jednotlivých kategorií lze nalézt v kapitole 11. V souladu se značením z kapitoly 4.3 označuje *nnz* počet nenulových prvků dané matice.

Všechny matice vybrané pro měření pocházejí z kolekce SuiteSparse Matrix Collection [24] a jsou tedy dostupné ke stažení na oficiálních stránkách kolekce na následující URL adrese: <https://sparse.tamu.edu/>

Všechny vybrané matice byly pro účely této práce rozděleny do složek podle názvu kategorie matice a byly zabaleny do jednoho archivu, který byl nahrán na úložiště ownCloud [82]. Tento archiv je veřejně dostupný na následující URL adrese: <https://owncloud.cesnet.cz/index.php/s/GaK9eHieKEv8CB9>

B.1 Velmi malé matice

Název	Řád	<i>nnz</i>	Datový typ
Pajek/GD96_a	1 096	1 677	integer
GHS_indef/exdata_1	6 001	2 269 500	real
Nemeth/nemeth23	9 506	1 506 810	real
Newman/netscience	1 589	5 484	real
Norris/heart3	2 339	680 341	real
Bai/qc2534	2 534	463 360	complex
HB/young1c	841	4 089	complex

■ **Tabulka B.1** Matice vybrané pro měření z kategorie velmi malých matic.

B.2 Malé matice

Název	Řád	<i>nnz</i>	Datový typ
Chen/pkustk12	94 653	7 512 317	pattern
Gupta/gupta3	16 783	9 323 427	pattern
Gaertner/pesa	11 738	79 566	integer
Bova/rma10	46 835	2 329 092	real
FEMLAB/poisson3Da	13 514	352 762	real
QY/case39	40 216	1 042 160	real
Williams/cant	62 451	4 007 383	real
Williams/consph	83 334	6 010 480	real
Williams/pdb1HYS	36 417	4 344 765	real
QCD/conf5_4-8x8-05	49 152	1 916 928	complex

■ **Tabulka B.2** Matice vybrané pro měření z kategorie malých matic.

B.3 Střední matice

Název	Řád	<i>nnz</i>	Datový typ
DIMACS10/ut2010	115 406	572 066	integer
Boeing/pwtk	217 918	11 524 432	real
DNVS/shipsec1	140 874	3 568 176	real
GHS_psdef/hood	220 542	9 895 422	real
Hamm/scircuit	170 998	958 936	real
PARSEC/SiO2	155 331	11 283 503	real
Oberwolfach/filter3D	106 437	2 707 179	real
Pajek/patents_main	240 547	560 943	real
Um/2cubes_sphere	101 492	1 647 264	real
Um/offshore	259 789	4 242 673	real
vanHeukelum/cage12	130 228	2 032 536	real
Williams/mac_econ_fwd500	206 500	1 273 389	real
Chevron/Chevron3	381 381	3 413 113	complex

■ **Tabulka B.3** Matice vybrané pro měření z kategorie středních matic.

B.4 Velké matice

Název	Řád	<i>nnz</i>	Datový typ
Bourchtein/atmosmodm	1 489 752	10 319 760	integer
Bodendiek/CurlCurl_3	1 219 574	13 544 618	real
Dziekonski/dielFilterV3real	1 102 824	89 306 020	real
Schenk/nlpkkt120	3 542 400	95 117 792	real
vanHeukelum/cage15	5 154 859	99 199 551	real
VLSI/vas_stokes_4M	4 382 246	131 577 616	real
Williams/webbase-1M	1 000 005	3 105 536	real

■ **Tabulka B.4** Matice vybrané pro měření z kategorie velkých matic.

B.5 Velmi velké matice

Název	Řád	<i>nnz</i>	Datový typ
DIMACS10/delaunay_n24	16 777 216	100 663 202	pattern
DIMACS10/hugetrace-00010	12 057 441	36 164 358	pattern
DIMACS10/rgg_n_2_24_s0	16 777 216	265 114 400	pattern
GAP/GAP-road	23 947 347	57 708 624	integer
FlowIPM22/Spielman_k400	21 413 602	64 401 604	real

■ **Tabulka B.5** Matice vybrané pro měření z kategorie velmi velkých matic.

Výsledky měření

Tato příloha obsahuje kompletní výsledky všech měření na fakultním klastru ClusterFIT, která byla provedena pro matice, jejichž detailní popis je uveden v příloze B.

Tato příloha je rozdělena do dvou hlavních částí. První část uvádí výsledky měření pro výpočetní uzly s procesorem Fujitsu A64FX, druhá část uvádí výsledky pro výpočetní uzel s procesorem AMD EPYC 7543P. Každá z těchto dvou hlavních částí je pak dále členěna podle kategorií matic, které byly vybrány pro měření. Bližší informace o výpočetních systémech použitých pro měření a popis kategorií vybraných řídkých matic jsou uvedeny v kapitole 11.

Pro každou kategorii matic jsou uvedeny tři tabulky. První tabulka uvádí čas SpGEMM výpočtu pro sekvenční variantu, paralelní OpenMP variantu a pro paralelní variantu s C++ vlákny. Druhá tabulka ukazuje pro paralelní OpenMP variantu paralelní výkonnostní metriky, konkrétně jsou uvedeny paralelní cena, paralelní zrychlení a paralelní efektivnost. Třetí tabulka pak uvádí tyto metriky pro paralelní variantu s C++ vlákny.

C.1 Výsledky pro uzly s procesorem Fujitsu A64FX

C.1.1 Velmi malé matice

Matrice	Sekvenční [s]	OpenMP [s]	C++ [s]
Pajek/GD96_a	0,002	0,001	0,016
GHS_indef/exdata_1	107,156	4,416	5,226
Nemeth/nemeth23	4,502	0,349	0,416
Newman/netscience	0,004	0,018	0,018
Norris/heart3	5,962	0,287	0,318
Bai/qc2534	1,805	0,149	0,209
HB/young1c	0,001	0,001	0,018

■ **Tabulka C.1** Časy výpočtu pro velmi malé matice na AArch64 uzlu.

Matrice	Cena	Zrychlení	Efektivnost
Pajek/GD96_a	0,048	2,000	0,041
GHS_indef/exdata_1	211,968	24,265	0,506
Nemeth/nemeth23	16,752	12,900	0,267
Newman/netscience	0,864	0,222	0,005
Norris/heart3	13,776	20,774	0,433
Bai/qc2534	7,152	12,114	0,252
HB/young1c	0,048	1,000	0,021

■ **Tabulka C.2** Výkonnostní metriky OpenMP varianty pro velmi malé matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Pajek/GD96_a	0,768	0,125	0,003
GHS_indef/exdata_1	250,848	20,504	0,427
Nemeth/nemeth23	19,968	10,822	0,225
Newman/netscience	0,864	0,222	0,005
Norris/heart3	15,264	18,748	0,391
Bai/qc2534	10,032	8,636	0,180
HB/young1c	0,864	0,055	0,001

■ **Tabulka C.3** Výkonnostní metriky varianty s C++ vlákny pro velmi malé matice na AArch64 uzlu.

C.1.2 Malé matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
Chen/pkustk12	109,732	18,138	17,462
Gupta/gupta3	1114,562	78,884	75,485
Gaertner/pesa	0,029	0,019	0,020
Bova/rma10	3,579	0,671	0,773
FEMLAB/poisson3Da	0,562	0,097	0,105
QY/case39	204,121	25,594	26,487
Williams/cant	6,016	0,755	0,789
Williams/consph	10,482	1,201	1,352
Williams/pdb1HYS	13,060	1,452	1,777
QCD/conf5_4-8x8-05	2,984	0,415	0,429

■ **Tabulka C.4** Časy výpočtu pro malé matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Chen/pkustk12	870,624	6,050	0,126
Gupta/gupta3	3786,432	14,129	0,294
Gaertner/pesa	0,912	1,526	0,031
Bova/rma10	32,208	5,334	0,111
FEMLAB/poisson3Da	4,656	5,794	0,121
QY/case39	1228,512	7,975	0,166
Williams/cant	36,240	7,968	0,166
Williams/consph	57,648	8,728	0,181
Williams/pdb1HYS	69,696	8,994	0,187
QCD/conf5_4-8x8-05	19,920	7,190	0,150

■ **Tabulka C.5** Výkonnostní metriky OpenMP varianty pro malé matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Chen/pkustk12	838,176	6,284	0,131
Gupta/gupta3	3623,280	14,765	0,307
Gaertner/pesa	0,960	1,450	0,030
Bova/rma10	37,104	4,630	0,096
FEMLAB/poisson3Da	5,040	5,352	0,112
QY/case39	1271,376	7,706	0,161
Williams/cant	37,872	7,625	0,159
Williams/consph	64,896	7,753	0,162
Williams/pdb1HYS	85,296	7,349	0,153
QCD/conf5_4-8x8-05	20,592	6,956	0,145

■ **Tabulka C.6** Výkonnostní metriky varianty s C++ vlákny pro malé matice na AArch64 uzlu.

C.1.3 Střední matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
DIMACS10/ut2010	0,966	0,069	0,112
Boeing/pwtk	14,720	1,450	1,549
DNVS/shipsec1	6,266	0,773	0,838
GHS_psdef/hood	14,048	1,371	1,507
Hamm/scircuit	4,740	0,242	0,285
PARSEC/SiO2	547,420	120,118	122,604
Oberwolfach/filter3D	3,702	0,579	0,635
Pajek/patents_main	2,577	0,123	0,171
Um/2cubes_sphere	1,518	0,236	0,238
Um/offshore	4,115	0,583	0,652
vanHeukelum/cage12	2,467	0,349	0,379
Williams/mac_econ_fwd500	1,055	0,122	0,172
Chevron/Chevron3	1,258	0,306	0,393

■ **Tabulka C.7** Časy výpočtu pro střední matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/ut2010	3,312	14,000	0,291
Boeing/pwtk	69,600	10,152	0,211
DNVS/shipsec1	37,104	8,106	0,168
GHS_psdef/hood	65,808	10,247	0,213
Hamm/scircuit	11,616	19,587	0,408
PARSEC/SiO2	5765,664	4,557	0,095
Oberwolfach/filter3D	27,792	6,394	0,133
Pajek/patents_main	5,904	20,951	0,436
Um/2cubes_sphere	11,328	6,432	0,134
Um/offshore	27,984	7,058	0,147
vanHeukelum/cage12	16,752	7,069	0,147
Williams/mac_econ_fwd500	5,856	8,648	0,180
Chevron/Chevron3	14,688	4,111	0,085

■ **Tabulka C.8** Výkonnostní metriky OpenMP varianty pro střední matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/ut2010	5,376	8,625	0,180
Boeing/pwtk	74,352	9,503	0,198
DNVS/shipsec1	40,224	7,477	0,155
GHS_psdef/hood	72,336	9,322	0,194
Hamm/scircuit	13,680	16,632	0,346
PARSEC/SiO2	5884,992	4,465	0,093
Oberwolfach/filter3D	30,480	5,830	0,121
Pajek/patents_main	8,208	15,070	0,314
Um/2cubes_sphere	11,424	6,378	0,133
Um/offshore	31,296	6,311	0,131
vanHeukelum/cage12	18,192	6,509	0,135
Williams/mac_econ_fwd500	8,256	6,134	0,127
Chevron/Chevron3	18,864	3,201	0,066

■ **Tabulka C.9** Výkonnostní metriky varianty s C++ vlákny pro střední matice na AArch64 uzlu.

C.1.4 Velké matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
Bourchtein/atmosmodm	4,331	0,816	1,013
Bodendiek/CurlCurl_3	8,252	1,435	1,687
Dziekonski/dielFilterV3real	292,654	39,332	40,099
Schenk/nlpkkt120	120,848	13,935	14,803
vanHeukelum/cage15	164,559	18,512	18,961
VLSI/vas_stokes_4M	346,003	33,938	34,838
Williams/webbase-1M	200,137	18,685	18,356

■ **Tabulka C.10** Časy výpočtu pro velké matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Bourchtein/atmosmodm	39,168	5,308	0,110
Bodendiek/CurlCurl_3	68,880	5,751	0,120
Dziekonski/dielFilterV3real	1887,936	7,441	0,155
Schenk/nlpkkt120	668,880	8,672	0,181
vanHeukelum/cage15	888,576	8,889	0,185
VLSI/vas_stokes_4M	1629,024	10,195	0,212
Williams/webbase-1M	896,880	10,711	0,223

■ **Tabulka C.11** Výkonnostní metriky OpenMP varianty pro velké matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Bourchtein/atmosmodm	48,624	4,275	0,089
Bodendiek/CurlCurl_3	80,976	4,892	0,102
Dziekonski/dielFilterV3real	1924,752	7,298	0,152
Schenk/nlpkkt120	710,544	8,164	0,170
vanHeukelum/cage15	910,128	8,678	0,180
VLSI/vas_stokes_4M	1672,224	9,932	0,207
Williams/webbase-1M	881,088	10,903	0,227

■ **Tabulka C.12** Výkonnostní metriky varianty s C++ vlákny pro velké matice na AArch64 uzlu.

C.1.5 Velmi velké matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
DIMACS10/delaunay_n24	88,701	7,810	10,488
DIMACS10/lugetrace-00010	14,572	2,225	4,486
DIMACS10/rgg_n_2_24_s0	207,303	28,786	28,997
GAP/GAP-road	21,234	4,083	7,896
FlowIPM22/Spielman_k400	984,610	24,269	27,900

■ **Tabulka C.13** Časy výpočtu pro velmi velké matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/delaunay_n24	374,880	11,357	0,237
DIMACS10/lugetrace-00010	106,800	6,549	0,136
DIMACS10/rgg_n_2_24_s0	1381,728	7,201	0,150
GAP/GAP-road	195,984	5,201	0,108
FlowIPM22/Spielman_k400	1164,912	40,571	0,845

■ **Tabulka C.14** Výkonnostní metriky OpenMP varianty pro velmi velké matice na AArch64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/delaunay_n24	503,424	8,457	0,176
DIMACS10/lugetrace-00010	215,328	3,248	0,067
DIMACS10/rgg_n_2_24_s0	1391,856	7,149	0,149
GAP/GAP-road	379,008	2,689	0,056
FlowIPM22/Spielman_k400	1339,200	35,291	0,735

■ **Tabulka C.15** Výkonnostní metriky varianty s C++ vlákny pro velmi velké matice na AArch64 uzlu.

C.1.6 Průměrné zrychlení

Skupina matic	OpenMP	C++
Velmi malé	10,468	8,445
Malé	7,369	6,987
Střední	9,793	8,112
Velké	8,138	7,735
Velmi velké	14,176	11,367

■ **Tabulka C.16** Průměrné zrychlení v rámci jednotlivých skupin matic na uzlu AArch64.

C.1.7 Časy výpočtu a zrychlení v závislosti na počtu vláken

Počet vláken	OpenMP [s]	C++ [s]
4	284,156	279,883
8	56,201	56,492
16	42,680	42,905
32	26,143	26,576
48	18,685	18,356

■ **Tabulka C.17** Časy výpočtu paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu AArch64.

Počet vláken	OpenMP	C++
4	0,704	0,715
8	3,561	3,543
16	4,689	4,665
32	7,655	7,531
48	10,711	10,903

■ **Tabulka C.18** Zrychlení paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu AArch64.

C.2 Výsledky pro uzel s procesorem AMD EPYC 7543P

C.2.1 Velmi malé matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
Pajek/GD96_a	0,0005	0,0002	0,018
GHS_indef/exdata_1	20,087	1,264	1,259
Nemeth/nemeth23	1,442	0,118	0,203
Newman/netscience	0,001	0,0004	0,021
Norris/heart3	1,245	0,082	0,082
Bai/qc2534	0,842	0,056	0,143
HB/young1c	0,0004	0,0004	0,019

■ **Tabulka C.19** Časy výpočtu pro velmi malé matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Pajek/GD96_a	0,013	2,500	0,039
GHS_indef/exdata_1	80,896	15,892	0,248
Nemeth/nemeth23	7,552	12,220	0,191
Newman/netscience	0,026	2,500	0,039
Norris/heart3	5,248	15,183	0,237
Bai/qc2534	3,584	15,036	0,235
HB/young1c	0,026	1,000	0,016

■ **Tabulka C.20** Výkonnostní metriky OpenMP varianty pro velmi malé matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Pajek/GD96_a	1,152	0,028	0,000
GHS_indef/exdata_1	80,576	15,955	0,249
Nemeth/nemeth23	12,992	7,103	0,111
Newman/netscience	1,344	0,048	0,001
Norris/heart3	5,248	15,183	0,237
Bai/qc2534	9,152	5,888	0,092
HB/young1c	1,216	0,021	0,000

■ **Tabulka C.21** Výkonnostní metriky varianty s C++ vlákny pro velmi malé matice na x86_64 uzlu.

C.2.2 Malé matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
Chen/pkustk12	29,081	8,194	8,299
Gupta/gupta3	266,115	15,460	15,426
Gaertner/pesa	0,011	0,004	0,028
Bova/rma10	1,152	0,264	0,291
FEMLAB/poisson3Da	0,210	0,044	0,090
QY/case39	52,701	9,669	9,694
Williams/cant	1,930	0,347	0,405
Williams/consph	3,412	0,532	0,577
Williams/pdb1HYS	3,828	0,542	0,610
QCD/conf5_4-8x8-05	1,090	0,197	0,298

■ **Tabulka C.22** Časy výpočtu pro malé matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Chen/pkustk12	524,416	3,549	0,055
Gupta/gupta3	989,440	17,213	0,269
Gaertner/pesa	0,256	2,750	0,043
Bova/rma10	16,896	4,364	0,068
FEMLAB/poisson3Da	2,816	4,773	0,075
QY/case39	618,816	5,451	0,085
Williams/cant	22,208	5,562	0,087
Williams/consph	34,048	6,414	0,100
Williams/pdb1HYS	34,688	7,063	0,110
QCD/conf5_4-8x8-05	12,608	5,533	0,086

■ **Tabulka C.23** Výkonnostní metriky OpenMP varianty pro malé matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Chen/pkustk12	531,136	3,504	0,055
Gupta/gupta3	987,264	17,251	0,270
Gaertner/pesa	1,792	0,393	0,006
Bova/rma10	18,624	3,959	0,062
FEMLAB/poisson3Da	5,760	2,333	0,036
QY/case39	620,416	5,436	0,085
Williams/cant	25,920	4,765	0,074
Williams/consph	36,928	5,913	0,092
Williams/pdb1HYS	39,040	6,275	0,098
QCD/conf5_4-8x8-05	19,072	3,658	0,057

■ **Tabulka C.24** Výkonnostní metriky varianty s C++ vlákny pro malé matice na x86_64 uzlu.

C.2.3 Střední matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
DIMACS10/ut2010	0,292	0,055	0,130
Boeing/pwtk	4,543	0,645	0,727
DNVS/shipsec1	2,000	0,392	0,473
GHS_psdef/hood	4,049	0,607	0,698
Hamm/scircuit	1,176	0,101	0,163
PARSEC/SiO2	104,846	28,594	28,712
Oberwolfach/filter3D	1,337	0,310	0,375
Pajek/patents_main	0,618	0,054	0,142
Um/2cubes_sphere	0,567	0,132	0,206
Um/offshore	1,556	0,346	0,433
vanHeukelum/cage12	0,818	0,209	0,270
Williams/mac_econ_fwd500	0,336	0,083	0,158
Chevron/Chevron3	0,499	0,186	0,373

■ **Tabulka C.25** Časy výpočtu pro střední matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/ut2010	3,520	5,309	0,083
Boeing/pwtk	41,280	7,043	0,110
DNVS/shipsec1	25,088	5,102	0,080
GHS_psdef/hood	38,848	6,670	0,104
Hamm/scircuit	6,464	11,644	0,182
PARSEC/SiO2	1830,016	3,667	0,057
Oberwolfach/filter3D	19,840	4,313	0,067
Pajek/patents_main	3,456	11,444	0,179
Um/2cubes_sphere	8,448	4,295	0,067
Um/offshore	22,144	4,497	0,070
vanHeukelum/cage12	13,376	3,914	0,061
Williams/mac_econ_fwd500	5,312	4,048	0,063
Chevron/Chevron3	11,904	2,683	0,042

■ **Tabulka C.26** Výkonnostní metriky OpenMP varianty pro střední matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/ut2010	8,320	2,246	0,035
Boeing/pwtk	46,528	6,249	0,098
DNVS/shipsec1	30,272	4,228	0,066
GHS_psdef/hood	44,672	5,801	0,091
Hamm/scircuit	10,432	7,215	0,113
PARSEC/SiO2	1837,568	3,652	0,057
Oberwolfach/filter3D	24,000	3,565	0,056
Pajek/patents_main	9,088	4,352	0,068
Um/2cubes_sphere	13,184	2,752	0,043
Um/offshore	27,712	3,594	0,056
vanHeukelum/cage12	17,280	3,030	0,047
Williams/mac_econ_fwd500	10,112	2,127	0,033
Chevron/Chevron3	23,872	1,338	0,021

■ **Tabulka C.27** Výkonnostní metriky varianty s C++ vlákny pro střední matice na x86_64 uzlu.

C.2.4 Velké matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
Bourchtein/atmosmodm	1,092	0,482	0,970
Bodendiek/CurlCurl_3	2,242	0,867	1,279
Dziekonski/dielFilterV3real	80,057	14,873	15,136
Schenk/nlpkkt120	32,912	8,106	8,768
vanHeukelum/cage15	52,194	12,053	13,700
VLSI/vas_stokes_4M	87,623	17,393	18,324
Williams/webbase-1M	55,331	4,301	4,645

■ **Tabulka C.28** Časy výpočtu pro velké matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Bourchtein/atmosmodm	30,848	2,266	0,035
Bodendiek/CurlCurl_3	55,488	2,586	0,040
Dziekonski/dielFilterV3real	951,872	5,383	0,084
Schenk/nlpkkt120	518,784	4,060	0,063
vanHeukelum/cage15	771,392	4,330	0,068
VLSI/vas_stokes_4M	1113,152	5,038	0,079
Williams/webbase-1M	275,264	12,865	0,201

■ **Tabulka C.29** Výkonnostní metriky OpenMP varianty pro velké matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
Bourchtein/atmosmodm	62,080	1,126	0,018
Bodendiek/CurlCurl_3	81,856	1,753	0,027
Dziekonski/dielFilterV3real	968,704	5,289	0,083
Schenk/nlpkkt120	561,152	3,754	0,059
vanHeukelum/cage15	876,800	3,810	0,060
VLSI/vas_stokes_4M	1172,736	4,782	0,075
Williams/webbase-1M	297,280	11,912	0,186

■ **Tabulka C.30** Výkonnostní metriky varianty s C++ vlákny pro velké matice na x86_64 uzlu.

C.2.5 Velmi velké matice

Matice	Sekvenční [s]	OpenMP [s]	C++ [s]
DIMACS10/delaunay_n24	28,781	6,625	10,607
DIMACS10/lugetrace-00010	5,726	2,392	5,432
DIMACS10/rgg_n_2_24_s0	79,342	15,805	19,914
GAP/GAP-road	8,164	5,042	9,668
FlowIPM22/Spielman_k400	234,409	13,183	18,530

■ **Tabulka C.31** Časy výpočtu pro velmi velké matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/delaunay_n24	424,000	4,344	0,068
DIMACS10/lugetrace-00010	153,088	2,394	0,037
DIMACS10/rgg_n_2_24_s0	1011,520	5,020	0,078
GAP/GAP-road	322,688	1,619	0,025
FlowIPM22/Spielman_k400	843,712	17,781	0,278

■ **Tabulka C.32** Výkonnostní metriky OpenMP varianty pro velmi velké matice na x86_64 uzlu.

Matice	Cena	Zrychlení	Efektivnost
DIMACS10/delaunay_n24	678,848	2,713	0,042
DIMACS10/lugetrace-00010	347,648	1,054	0,016
DIMACS10/rgg_n_2_24_s0	1274,496	3,984	0,062
GAP/GAP-road	618,752	0,844	0,013
FlowIPM22/Spielman_k400	1185,920	12,650	0,198

■ **Tabulka C.33** Výkonnostní metriky varianty s C++ vlákny pro velmi velké matice na x86_64 uzlu.

C.2.6 Průměrné zrychlení

Skupina matic	OpenMP	C++
Velmi malé	9,190	6,318
Malé	6,267	5,349
Střední	5,741	3,858
Velké	5,218	4,632
Velmi velké	6,232	4,249

■ **Tabulka C.34** Průměrné zrychlení v rámci jednotlivých skupin matic na uzlu x86_64.

C.2.7 Časy výpočtu a zrychlení v závislosti na počtu vláken

Počet vláken	OpenMP [s]	C++ [s]
4	72,385	70,086
8	15,247	15,123
16	12,212	12,916
32	7,948	7,943
48	5,878	6,199
64	4,301	4,645

■ **Tabulka C.35** Časy výpočtu paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu x86_64.

Počet vláken	OpenMP	C++
4	0,764	0,789
8	3,629	3,659
16	4,531	4,284
32	6,962	6,966
48	9,413	8,926
64	12,865	11,912

■ **Tabulka C.36** Zrychlení paralelních variant v závislosti na počtu vláken pro matici Williams/webbase-1M na uzlu x86_64.

[illegible]

Další dokumenty

Tato příloha obsahuje formální dokumenty vyžadované Metodickým pokynem č. 6/2023 pro zadávání, odevzdávání, ukládání a zpřístupňování bakalářských a diplomových závěrečných prací. Konkrétně tato příloha obsahuje dva dokumenty vygenerované z univerzitního informačního systému KOS. Prvním dokumentem je zadání této diplomové práce a druhým dokumentem je prohlášení o vypracování závěrečné práce.



ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Javorka** Jméno: **Matěj** Osobní číslo: **372565**
Fakulta/ústav: **Fakulta informačních technologií**
Zadávající katedra/ústav: **Katedra softwarového inženýrství**
Studijní program: **Informatika**
Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vyhodnocení paralelizačních přístupů pro efektivní násobení řídkých matic

Název diplomové práce anglicky:

Evaluation of parallelization approaches for efficient multiplication of sparse matrices

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Ivan Šimeček, Ph.D. katedra počítačových systémů FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **16.09.2025**

Termín odevzdání diplomové práce: _____

podpis vedoucí(ho) ústavu/katedry

podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Bc. Javorka Matěj

Podpis studenta

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Javorka** Jméno: **Matěj** Osobní číslo: **372565**
Fakulta/ústav: **Fakulta informačních technologií**
Zadávající katedra/ústav: **Katedra softwarového inženýrství**
Studijní program: **Informatika**
Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vyhodnocení paralelizačních přístupů pro efektivní násobení řídkých matic

Název diplomové práce anglicky:

Evaluation of parallelization approaches for efficient multiplication of sparse matrices

Pokyny pro vypracování:

- 1) Seznamte se s problematikou ukládání řídkých matic a analyzujte existující formáty pro jejich ukládání. [1-3]
 - 2) Nastudujte a analyzujte algoritmy pro násobení řídkých matic jak v základních, tak pokročilých formátech jako ELL či BCRS. [1-7]
 - 3) Pro alespoň 2 různé formáty implementujte v C++ sekvenční algoritmus pro násobení řídkých matic a také jeho paralelní verzi pomocí obou následujících různých paralelizačních přístupů:
 - a) C++ bez externích knihoven pro paralelizaci;
 - b) s použitím externí knihovny pro paralelizaci nad sdílenou pamětí.
 - 4) Při návrhu dbejte na znovupoužitelnost, udržitelnost a snadnou rozšiřitelnost kódu. Prostudujte a diskutujte možnosti využití moderních vlastností jazyka ze standardu C++20. [8]
 - 5) Všechna implementovaná řešení řádně otestujte a zautomatizujte kompilaci i ověření testů pomocí kontinuální integrace.
 - 6) Změřte dobu běhu, pro paralelní řešení pak navíc i paralelní cenu, zrychlení a efektivnost. Porovnejte a diskutujte výkon jednotlivých implementovaných řešení.
- Seznam odborné literatury:
- [1] SCOTT, Jennifer a TÚMA, Miroslav. Algorithms for Sparse Linear Systems. Springer Nature, 2023. ISBN 978-3-031-25820-6.
 - [2] DUFF, Iain S.; ERISMAN, A. M. a REID, John Ker. Direct methods for sparse matrices / I.S. Duff, A.M. Erisman, J.K. Reid. Numerical mathematics and scientific computation (Oxford). Oxford: Clarendon Press, c1986. ISBN 978-0198534211.
 - [3] OSTERBY, Ole a ZLATEV, Zahari. Direct Methods for Sparse Matrices. Lecture notes in computer science. Berlin: Springer-Verlag, 1983. ISBN 978-3-540-12676-8.
 - [4] JERZY S. RESPONDEK. Fast Matrix Multiplication with Applications. Springer Nature, 2025. ISBN 978-3-031-76930-6.
 - [5] YUSTER, Raphael a Uri ZWICK. Fast sparse matrix multiplication. ACM transactions on algorithms. 2005, roč. 1, č. 1, s. 2-13.
 - [6] HONG, Chuhe; WANG, Qinglin; MAO, Runzhang; LIANG, Yuechao; XIA, Rui et al. SaSpGEMM: Sorting-Avoiding Sparse General Matrix-Matrix Multiplication on Multi-Core Processors. Online. In: Proceedings of the 53rd International Conference on Parallel Processing (ICPP '24). 2024, s. 1166-1175. Dostupné z: <https://doi.org/10.1145/3673038.3673054>.
 - [7] WOLFSON-POU, Jordi; LAUKEMANN, Jan a PETRINI, Fabrizio. MAGNUS: Generating Data Locality to Accelerate Sparse Matrix-Matrix Multiplication on CPUs. Online. In: Proceedings of the 39th ACM International

Conference on Supercomputing (ICS '25). 2025, s. 442-457. Dostupné z:
<https://doi.org/10.1145/3721145.3725773>.
[8] NICOLAI M. JOSUTTIS. C++ 20: The Complete Guide. 2022. ISBN 978-3-96730-020-8.

Seznam doporučené literatury:

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY
Thákurova 9
160 00 Praha 6



PROHLÁŠENÍ

Já, níže podepsaný

Příjmení, jméno studenta: Javorka Matěj
Osobní číslo: 372565
Název programu: Informatika

prohlašuji, že jsem diplomovou práci s názvem

Vyhodnocení paralelizačních přístupů pro efektivní násobení řídkých matic

vypracoval samostatně a uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací a Rámcovými pravidly používání umělé inteligence na ČVUT pro studijní a pedagogické účely v Bc a NM studiu.

Prohlašuji, že jsem v průběhu příprav a psaní závěrečné práce použil nástroje umělé inteligence. Vygenerovaný obsah jsem ověřil. Stvrzuji, že jsem si vědom, že za obsah závěrečné práce plně zodpovídám.

V Praze dne 06.05.2026

Bc. Matěj Javorka

.....
podpis studenta

Bibliografie

1. GAO, Jianhua; JI, Weixing; CHANG, Fangli; SHI-YU, Han; WEI, Bingxin; LIU, Zeming; WANG, Yizhuo. A Systematic Survey of General Sparse Matrix-matrix Multiplication. *ACM Computing Surveys*. 2023, roč. 55, č. 12, s. 1–36. ISSN 0360-0300. Dostupné z DOI: 10.1145/3571157.
2. PATTERSON, David A.; HENNESSY, John L. *Computer Organization and Design: The Hardware/Software Interface*. 4th rev. ed. Waltham: Morgan Kaufmann, 2012. ISBN 978-0-12-374750-1.
3. HENNESSY, John L.; PATTERSON, David A. *Computer Architecture: A Quantitative Approach*. 6th ed. Cambridge, MA: Morgan Kaufmann Publishers, Elsevier, 2019. ISBN 978-0-12-811905-1.
4. HARRIS, David Money; HARRIS, Sarah L. *Digital Design and Computer Architecture*. 2nd ed. Waltham: Morgan Kaufmann, 2012. ISBN 978-0-12-394424-5.
5. WOLFSON-POU, Jordi; LAUKEMANN, Jan; PETRINI, Fabrizio. MAGNUS: Generating Data Locality to Accelerate Sparse Matrix-Matrix Multiplication on CPUs. In: *ICS '25: Proceedings of the 39th ACM International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2025, sv. 2025, s. 442–457. Č. 2025. ISBN 979-8-4007-1537-2. Dostupné z DOI: 10.1145/3721145.3725773.
6. SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. *Operating System Concepts*. 10th ed. John Wiley & Sons, 2018. ISBN 978-1-119-43925-7.
7. STALLINGS, William. *Operating Systems: Internals and Design Principles*. 7th ed. Upper Saddle River: Pearson Education, 2012. ISBN 978-0-13-230998-1.
8. TANENBAUM, Andrew S.; BOS, Herbert. *Modern Operating Systems*. 4th ed. Boston: Pearson, 2015. ISBN 978-0-13-359162-0.

9. FUJITSU. *Fujitsu A64FX: Technical Presentation*. 2025. Dostupné také z: https://epicure-hpc.eu/wp-content/uploads/2025/03/A64FX_Technical_presentation_epicure-Feb2025.pdf. cit. 2026-05-01.
10. MATTSON, Timothy G.; SANDERS, Beverly A.; MASSINGILL, Berna L. *Patterns for Parallel Programming*. Boston: Addison-Wesley, 2005. ISBN 0-321-22811-1.
11. GRIMM, Rainer. *Concurrency with Modern C++*. Independently published, 2024. ISBN 979-8-8734-2093-3.
12. STROUSTRUP, Bjarne. *The C++ Programming Language*. 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 978-0-321-56384-2.
13. JOSUTTIS, Nicolai. *C++20: The Complete Guide*. Nicojosuttis, 2022. ISBN 978-3-96730-920-1.
14. OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Programming Interface Specification Version 5.0*. Ed. KLEMM, Michael; SUPINSKI, Bronis R. Independently published, 2019. ISBN 978-1-7957-5988-5.
15. CHAPMAN, Barbara; JOST, Gabriele; PAS, Ruud van der. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge: The MIT Press, 2008. ISBN 978-0-262-53302-7.
16. LEE, Jooeun; OH, Seong Eun; HONG, Jung-Wuk. Parallel Programming of a Peridynamics Code Coupled with Finite Element Method. *International Journal of Fracture*. 2017, roč. 203, č. 1-2, s. 99–114. ISSN 0376-9429. Dostupné z DOI: 10.1007/s10704-016-0121-y.
17. MATOUŠEK, Jiří; NEŠETŘIL, Jaroslav. *Kapitoly z diskrétní matematiky*. 4. upr. a dopl. vyd. Praha: Univerzita Karlova, nakladatelství Karolinum, 2019. ISBN 978-80-246-1740-4.
18. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. 2. vyd. Praha: CZ.NIC, z.s.p.o., 2022. ISBN 978-80-88168-63-8.
19. MILLER, Russ; BOXER, Laurence. *Algorithms Sequential and Parallel: A Unified Approach*. 3rd ed. Boston, MA: Cengage Learning, 2013. ISBN 978-1-133-36680-5.
20. OLŠÁK, Petr. *Úvod do algebry, zejména lineární*. 2. vyd. Praha: Česká technika – nakladatelství ČVUT, 2013. ISBN 978-80-01-05291-4.
21. BOYD, Stephen; VANDENBERGHE, Lieven. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018. ISBN 978-1-316-51896-0.
22. HORN, Roger A.; JOHNSON, Charles R. *Matrix Analysis*. 2nd ed. Cambridge: Cambridge University Press, 2013. ISBN 978-0-521-54823-6.

23. DAVIS, Timothy A. *Direct Methods for Sparse Linear Systems*. Philadelphia: Society for Industrial and Applied Mathematics, 2006. ISBN 978-0-898716-13-9.
24. DAVIS, Timothy A.; HU, Yifan. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*. 2011, roč. 38, č. 1, s. 1–25. ISSN 0098-3500. Dostupné z DOI: 10.1145/2049662.2049663.
25. DEMEK, Miroslav. *marc*. 2022. Dostupné také z: <https://github.com/bigno78/marc>.
26. SCOTT, Jennifer; TŮMA, Miroslav. *Algorithms for Sparse Linear Systems*. Cham: Birkhäuser, 2023. ISBN 978-3-031-25819-0.
27. ØSTERBY, Ole; ZLATEV, Zahari. Direct Methods for Sparse Matrices. *DAIMI Report Series*. 1980, roč. 9, č. 123. ISSN 0105-8517. Dostupné z DOI: 10.7146/dpb.v9i123.6541.
28. GOLUB, Gene H.; LOAN, Charles F. Van. *Matrix Computations*. 4th ed. Baltimore, Md.: Johns Hopkins University Press, 2013. ISBN 978-1-4214-0794-4.
29. DUFF, Iain S.; ERISMAN, Albert M.; REID, John K. *Direct Methods for Sparse Matrices*. 2nd ed. New York: Oxford University Press, 2017. ISBN 978-0-19-850838-0.
30. YUSTER, Raphael; ZWICK, Uri. Fast Sparse Matrix Multiplication. *ACM Transactions on Algorithms*. [B.r.], roč. 1, č. 1, s. 2–13. ISSN 1549-6325. Dostupné z DOI: 10.1145/1077464.1077466.
31. FILIPPONE, Salvatore; CARDELLINI, Valeria; FANFARILLO, Alessandro; BARBIERI, Davide. Sparse Matrix-Vector Multiplication on GP-GPUs. *ACM Transactions on Mathematical Software*. 2017, roč. 43, č. 4, s. 1–49. ISSN 0098-3500. Dostupné z DOI: 10.1145/3017994.
32. GU, Zhixiang; MOREIRA, Jose; EDELSON, David; AZAD, Ariful. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking. In: *SPAA '20*. Virtual Event, USA: Association for Computing Machinery, 2020, s. 293–303. ISBN 978-1-4503-6935-0. Dostupné z DOI: 10.1145/3350755.3400216.
33. SRIVASTAVA, Nitish; JIN, Hanchen; LIU, Jie; ZHANG, Zhiru; ALBONESI, David. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In: 2020, s. 766–780. Dostupné z DOI: 10.1109/micro50266.2020.00068.
34. ZHANG, Guowei; ATTALURI, Nithya; EMER, Joel S.; SÁNCHEZ, Daniel. GAMMA: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In: *ASPLOS '21: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for

- Computing Machinery, 2021. ISBN 978-1-4503-8317-2. Dostupné z DOI: 10.1145/3445814.3446702.
35. SAAD, Yousef. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics, 2003. ISBN 0-89871-534-2. Dostupné z DOI: 10.1137/1.9780898718003.
 36. GAO, Jianhua; LIU, Bingjie; JI, Weixing; HUANG, Hua. A Systematic Literature Survey of Sparse Matrix-Vector Multiplication. *ArXiv*. 2024. Dostupné z DOI: 10.48550/arxiv.2404.06047.
 37. BELL, Nathan; GARLAND, Michael. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, s. 1–11. Dostupné z DOI: 10.1145/1654059.1654078.
 38. SHAHNAZ, Rukhsana; USMAN, Anila; CHUGHTAI, Imran Rafiq. Review of Storage Techniques for Sparse Matrices. In: *Pakistan Section Multitopic Conference*. 2005, s. 1–7. Dostupné z DOI: 10.1109/inmic.2005.334453.
 39. VUDUC, Richard W.; MOON, Hyun-Jin. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In: *High Performance Computing and Communications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, s. 807–816. ISBN 978-3-540-32079-1. Dostupné z DOI: 10.1007/11557654_91.
 40. LIU, Weifeng; VINTER, Brian. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. *ArXiv*. 2015. Dostupné z DOI: 10.48550/arxiv.1503.05032.
 41. DUFF, Iain S.; GRIMES, Roger G.; LEWIS, John G. Sparse Matrix Test Problems. *ACM Transactions on Mathematical Software*. 1989, roč. 15, č. 1, s. 1–14. ISSN 0098-3500. Dostupné z DOI: 10.1145/62038.62043.
 42. BOISVERT, Ronald F.; REMINGTON, Karin; BARRETT, Richard F.; POZO, Roldan; DONGARRA, Jack J. Matrix Market: A Web Resource for Test Matrix Collections. In: *Quality of Numerical Software: Assessment and enhancement*. Boston, MA: Springer US, 1997, s. 125–137. ISBN 978-1-5041-2940-4. ISSN 1868-422X. Dostupné z DOI: 10.1007/978-1-5041-2940-4_9.
 43. DUFF, Iain S.; GRIMES, Roger G.; LEWIS, John G. The Rutherford-Boeing Sparse Matrix Collection. *Technical report. RAL - Council for the Central Laboratory of the Research Councils*. 1997, roč. 86, č. 10. ISSN 1358-6254.
 44. BOISVERT, Ronald; POZO, Roldan; REMINGTON, Karin. The Matrix Market Exchange Formats: Initial Design. *NIST Interagency/Internal Report*. 1996. Dostupné také z: <https://www.nist.gov/publications/matrix-market-exchange-formats-initial-design>.

45. HONG, Chuhe; WANG, Qinglin; MAO, Runzhang; LIANG, Yuechao; XIA, Rui; LIU, Jie. SaSpGEMM: Sorting-Avoiding Sparse General Matrix-Matrix Multiplication on Multi-Core Processors. *Proceedings of the 53rd International Conference on Parallel Processing*. 2024, s. 1166–1175. Dostupné z DOI: 10.1145/3673038.3673054.
46. DALTON, Steven; OLSON, Luke N.; BELL, Nathan. Optimizing Sparse Matrix-Matrix Multiplication for the GPU. *ACM Transactions on Mathematical Software*. 2015, roč. 41, č. 4. ISSN 0098-3500. Dostupné z DOI: 10.1145/2699470.
47. LIU, Weifeng; VINTER, Brian. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing*. 2015, roč. 85, s. 47–61. ISSN 0743-7315. Dostupné z DOI: 10.1016/j.jpdc.2015.06.010.
48. DU, Zhaoyang; GUAN, Yijin; GUAN, Tianchan; NIU, Dimin; ZHENG, Hongzhong; XIE, Yuan. Accelerating CPU-Based Sparse General Matrix Multiplication With Binary Row Merging. *IEEE Access*. 2022, roč. 10, s. 79237–79248. ISSN 2169-3536. Dostupné z DOI: 10.1109/access.2022.3193937.
49. GREMSE, Felix; HÖFTER, Andreas; SCHWEN, Lars Ole; KIESSLING, Fabian; NAUMANN, Uwe. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing*. 2015, roč. 37, č. 1, s. C54–C71. ISSN 1064-8275. Dostupné z DOI: 10.1137/130948811.
50. COHEN, Edith. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization*. 1998, roč. 2, č. 4, s. 307–332. ISSN 1382-6905. Dostupné z DOI: 10.1023/A:1009716300509.
51. AMOSSEN, Rasmus Resen; CAMPAGNA, Andrea; PAGH, Rasmus. Better Size Estimation for Sparse Matrix Products. *Algorithmica*. 2014, roč. 69, č. 3, s. 741–757. ISSN 0178-4617. Dostupné z DOI: 10.1007/s00453-012-9692-9.
52. ANH, Pham Nguyen Quang; FAN, Rui; WEN, Yonggang. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. *Proceedings of the 2016 International Conference on Supercomputing*. 2016. Dostupné z DOI: 10.1145/2925426.2926273.
53. GILBERT, John R.; MOLER, Cleve; SCHREIBER, Robert. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*. 1992, roč. 13, č. 1, s. 333–356. ISSN 0895-4798. Dostupné z DOI: 10.1137/0613024.

54. AZAD, Ariful; BALLARD, Grey; BULUÇ, Aydın; DEMMEL, James; GRIGORI, Laura; SCHWARTZ, Oded; TOLEDO, Sivan; WILLIAMS, Samuel. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. *SIAM Journal on Scientific Computing*. 2016, roč. 38, č. 6, s. C624–C651. ISSN 1095-7197. Dostupné z DOI: 10.1137/15m104253x.
55. BALLARD, Grey; BULUÇ, Aydın; DEMMEL, James; GRIGORI, Laura; LIPSHITZ, Benjamin; SCHWARTZ, Oded; TOLEDO, Sivan. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: Association for Computing Machinery, 2013, s. 222–231. ISBN 978-1-4503-1572-2. Dostupné z DOI: 10.1145/2486159.2486196.
56. BALLARD, Grey; DRUINSKY, Alex; KNIGHT, Nicholas; SCHWARTZ, Oded. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *ACM Transactions on Parallel Computing*. 2016, roč. 3, č. 3. ISSN 2329-4949. Dostupné z DOI: 10.1145/3015144.
57. PADUA, David A. *Encyclopedia of Parallel Computing*. New York, NY: Springer, 2014. ISBN 978-0-387-09766-4. Dostupné z DOI: 978-0-387-09766-4.
58. AKBUDAK, Kadir; AYKANAT, Cevdet. Simultaneous Input and Output Matrix Partitioning for Outer-Product-Parallel Sparse Matrix-Matrix Multiplication. *SIAM Journal on Scientific Computing*. 2014, roč. 36, č. 5, s. C568–C590. ISSN 1064-8275. Dostupné z DOI: 10.1137/13092589x.
59. GUSTAVSON, Fred G. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software*. 1978, roč. 4, č. 3, s. 250–269. ISSN 0098-3500. Dostupné z DOI: 10.1145/355791.355796.
60. NAGASAKA, Yusuke; MATSUOKA, Satoshi; AZAD, Ariful; BULUÇ, Aydın. Performance Optimization, Modeling and Analysis of Sparse Matrix-Matrix Products on Multi-Core and Many-Core Processors. *Parallel Computing*. 2019, č. 90. ISSN 0167-8191. Dostupné z DOI: 10.1016/j.parco.2019.102545.
61. BRAMAS, Béranger. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications*. 2017, roč. 8, č. 10. ISSN 2156-5570. Dostupné z DOI: 10.14569/ijacsa.2017.081044.
62. *CPlusPlus* [<https://cplusplus.com/>]. 2026. cit. 2026-02-17.
63. *C++ reference* [<https://en.cppreference.com/>]. 2026. cit. 2026-02-17.

64. KNUTH, Donald Ervin. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. ed. Addison–Wesley, 1998. ISBN 978-0-201-89685-5.
65. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd. ed. Cambridge: MIT Press, 2009. ISBN 978-0-262-03384-8.
66. INC., Wolfram Research. *Wolfram/Alpha Notebook Edition*. 2026. Dostupné také z: <https://www.wolfram.com/wolfram-alpha-notebook-edition>. Champaign, IL, 2026.
67. SCHACH, Stephen R. *Object-Oriented and Classical Software Engineering*. 8th. ed. New York: McGraw-Hill, 2011. ISBN 978-0-07-337618-9.
68. LUGOWSKI, Adam. *fast_matrix_market: Fast and Full-Featured Matrix Market I/O Library*. 2023. Dostupné také z: https://github.com/alugowski/fast_matrix_market. cit. 2026-02-22.
69. *OpenMP Compilers & Tools* [<https://www.openmp.org/resources/openmp-compilers-tools/>]. 2025. cit. 2026-03-01.
70. MCCOOL, Michael; REINDERS, James; ROBISON, Arch D. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012. ISBN 978-0-12-415993-8.
71. IEEE. *754-2019 - IEEE Standard for Floating-Point Arithmetic*. 2019. Dostupné z DOI: 10.1109/IEEESTD.2019.8766229.
72. P-RANAV. *argparse – Argument Parser for Modern C++*. 2025. Dostupné také z: <https://github.com/p-ranav/argparse>. cit. 2026-02-22.
73. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis: Addison-Wesley, 1995. ISBN 0-201-63361-2.
74. KIRILOV, Viktor. *Doctest*. 2025. Dostupné také z: <https://github.com/doctest/doctest>. cit. 2026-03-13.
75. HOŘEŇOVSKÝ, Martin. *Catch2*. 2026. Dostupné také z: <https://github.com/catchorg/Catch2>. cit. 2026-03-13.
76. *CMake*. 2026. Dostupné také z: <https://cmake.org/>. cit. 2026-03-16.
77. *Make*. 2026. Dostupné také z: <https://www.gnu.org/software/make/>. cit. 2026-03-16.
78. *Ninja*. 2026. Dostupné také z: <https://ninja-build.org/>. cit. 2026-03-16.
79. *Docker Hub*. 2026. Dostupné také z: <https://hub.docker.com/>. cit. 2026-03-18.
80. *PagesFIT*. 2026. Dostupné také z: <https://help.fit.cvut.cz/pages-fit/index.html>. cit. 2026-03-18.

81. AMD. *AMD EPYC 7543P*. 2026. Dostupné také z: <https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7543p.html>. cit. 2026-05-01.
82. CESNET. *ownCloud*. 2026. Dostupné také z: <https://owncloud.cesnet.cz/>. cit. 2026-03-30.
83. GUENNEBAUD, Gaël; JACOB, Benoît et al. *Eigen*. 2010. Dostupné také z: <https://libeigen.gitlab.io>. cit. 2026-05-05.
84. WILLIAMS, Samuel; WATERMAN, Andrew; PATTERSON, David. Ro-offline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*. 2009, roč. 52, č. 4, s. 65–76. ISSN 0001-0782. Dostupné z DOI: 10.1145/1498765.1498785.
85. DU, Zhaoyang; GUAN, Yijin; GUAN, Tianchan; NIU, Dimin; HUANG, Linyong; ZHENG, Hongzhong; XIE, Yuan. OpSparse: A Highly Optimized Framework for Sparse General Matrix Multiplication on GPUs. *IEEE Access*. 2022, roč. 10, s. 85960–85974. Dostupné z DOI: 10.1109/ACCESS.2022.3196940.

Obsah přiloženého archivu

```
/
├── readme.txt ..... stručný popis obsahu archivu
├── src
│   ├── implementation.zip ..... zdrojové kódy implementace
├── text ..... text práce
│   ├── thesis.zip ..... zdrojové kódy práce ve formátu LATEX
│   └── thesis.pdf ..... text práce ve formátu PDF
```