VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

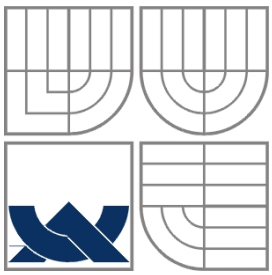
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KNIHOVNA OPERACÍ NAD KONEČNÝMI AUTOMATY

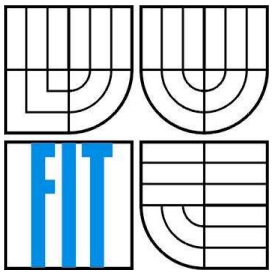
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PETR BARTŮNĚK
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KNIHOVNA OPERACÍ NAD KONEČNÝMI AUTOMATY

LIBRARY FOR OPERATIONS OVER FINITE AUTOMATA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR BARTŮNĚK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KAŠTIL

BRNO 2010

Abstrakt

Tato práce se zabývá dvěma základními operacemi nad konečnými automaty. Determinizací nedeterministických konečných automatů a minimalizací deterministických konečných automatů. Pro obě tyto operace jsem navrhoval sekvenční algoritmy, které jsou paralelizovatelné. Zabývám se hledáním zrychlení především pomocí SSE instrukcí nebo pomocí knihovny openMP. Trendem dnešní doby je především zvyšování počtu procesorů, proto budu navrhovat paralelní algoritmy pro více procesorů. Při hledání optimálního řešení budu zkoumat další možnosti, jak dosáhnout zrychlení, např. efektivním uložením datových struktur v paměti.

Abstract

This work deals with two basic operations over finite automata. Determination of nondeterministic finite automata and minimization of deterministic finite automata. For these two operations I proposed sequential algorithms that are parallelizable. I deal mainly with finding the speedup of SSE instructions, or use the OpenMP library. The trend today is mainly in increasing the number of processors, so I propose parallel algorithms for multiple processors. When searching for the optimal solution, I will be to examine other ways to achieve speedup, for example efficient saving of the data structures in memory.

Klíčová slova

Konečný automat, determinizace, minimalizace, optimalizace, paralelní programování, SSE, OpenMP

Keywords

Finite automata, determination, minimization, optimization, parallel programming, SSE, OpenMP

Citace

Bartůněk Petr: Knihovna operací nad konečnými automaty, Brno, FIT VUT v Brně, 2010.

Knihovna operací nad konečnými automaty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……..
Petr Bartůněk
26.5. 2010

Poděkování

Děkuji panu Ing. Janu Kaštilovi za jeho odborné vedení projektu, dále děkuji všem, kteří mi poskytli pomoc při řešení projektu, jmenovitě je to především můj přítel Bc. David Čeloud.

Obsah

1 Úvod

Protože často pracujeme se složitými konečnými automaty, a operace nad nimi jsou velmi časově náročné, snažíme se nalézt rychlejší algoritmy pro jejich zpracování. V dnešní době se již nezvyšují tolik rychlosti procesorů, nýbrž se zvětšuje jejich počet. V dnešní době je velkým trendem zvyšování počtu procesorů v počítači. Je to prozatím největší potenciál zrychlení. Proto navrhujeme algoritmy, které můžou běžet paralelně na více procesorech současně a urychlují tak celý výpočet.

Cílem této práce je tedy prozkoumat problematiku dvou základních operací nad konečnými automaty. Jedná se o determinizaci nedeterministických konečných automatů a minimalizací deterministických konečných automatů. Zkoumám jejich paralelní implementaci a snažím se nalézt optimální rychlé řešení těchto problémů.

Ve druhé kapitole jsou nejdříve vysvětleny základní pojmy týkající se konečných automatů obecně. Čtenář získá přehled o základních formálních algoritmech obou operací. Tyto znalosti jsou základní a nedalo by se bez nich obejít.

Třetí kapitola je také teoretická a zabývá se paralelním programováním. Je zde probrána Flynnova klasifikace paralelních strojů. Dále zde můžeme naleznout důležité modely paralelního programování. Tyto modely jsou důležité, protože bude potřeba se rozhodnout a zvolit pro každý konkrétní algoritmus vhodný model, aby implementace paralelní verze byla co nejefektivnější. Zde se jedná především o paralelní programování se sdílenou pamětí nebo zasíláním zpráv. Za zmínku stojí i paralelní a hybridní model dat. Detailněji se však zaměřím na programování s využitím knihovny openMP, kterou budu v programu využívat. Jedná se o model se sdílenou pamětí.

Ve čtvrté kapitole je možné nalézt krátký teoretický úvod k hardwarovým architekturám počítačů. Je dobré vědět, co nám dnešní výpočetní stroje dovolují nebo jak mohou být postavené.

Před samotnou implementací knihovny se ještě krátce zmíním v kapitole 5 o profilovacích nástrojích, které můžeme využívat při programování, které nám pomáhají ladit programy a hledat chyby. Nejznámějšími nástroji jsou valgrind a GNU GPROF.

V kapitole 6 je popsána práce na knihovně. Jsou zde popsány a podrobně rozebrány zkoumané algoritmy.

Při hledání zrychlení jsem navíc provedl test SSE instrukcí, kdy jsem zkoumal rychlost standardních funkcí jazyka C a funkcí napsaných v SSE. SSE nám umožní vykonávat instrukce SIMD (Single instruction – multiple data), kdy my zadáme jednu instrukci, která se provede nad několika daty současně v jednom kroku. Takto můžeme provést porovnání dvou velkých vektorů nebo jejich vzájemné sečtení, odečtení, bitové operace s nimi a další. O SSE a jeho efektivitě či vhodnosti ho používat bude ještě jedna kapitola. Tam se dozvíme, pro které případy bude vhodné používat SSE instrukce, a kde je zase naopak výhodnější použít optimalizované knihovní funkce jazyka C.

Dále následuje návrh datových struktur jednotlivých automatů. Je velmi důležité, jak budeme automaty v paměti ukládat, to se může projevit na celkové rychlosti běhu algoritmu. Jsou navrženy datové struktury NFA a DFA. Dále jsem navrhnul způsob uložení těchto struktur do souboru (pro budoucí použití automatů). Pro determinizaci jsem navrhnul jeden sekvenční algoritmus a k němu hned tři verze paralelních algoritmů. První verze používala SSE instrukce a další dvě verze byly naprogramovány pomocí openMP. Pro minimalizaci jsem navrhoval hned dvě rozdílné sekvenční verze. Pro každou jsem napsal i její paralelní verzi. Všechny verze všech algoritmů jsem mezi sebou porovnával a zjišťoval jejich vhodnost. U paralelních verzí jsem zjišťoval zrychlení na více procesorech a efektivitu řešení.

Na konci této publikace se ještě zmíním o praktickém využití knihovny. Popíšu zde demonstrační program, který jsem naprogramoval pro účely testování.

2 Konečné automaty

Konečný automat (KA), anglickým názvem finite automaton (FA) je základním výpočetním modelem používaným v informatice především v teorii vyčíslitelnosti a obecně v teorii formálních jazyků. Reprezentuje jednoduchý výpočetní stroj, který se vždy nachází v jednom ze svých stavů. Z těchto stavů přechází pomocí přechodových funkcí do stavů následujících při současném čtení vstupních symbolů, takto se stává akceptorem regulárních jazyků. Množina stavů je konečná, odtud je i jeho název. Na rozdíl od jiných automatů, konečný automat nemá žádnou další paměť, automat zná pouze aktuální stav, ve kterém se nachází a přechodové funkce pro všechny stavy. Protože je konečný automat velmi jednoduchý, nedokáže přijímat jiné než regulární jazyky. Konečný automat používáme při zpracovávání regulárních výrazů např. v lexikálním analyzátoru v překladači. [1]

2.1 Formální definice konečného automatu

Nedeterministický konečný automat je definován jako uspořádaná pětice $M = (Q, \Sigma, \delta, s, F)$, kde:

- Q je konečná množina všech stavů.
- Σ je konečná množina všech vstupních symbolů, nazýváme ji vstupní abeceda.
- δ je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi jednotlivými stavy. U nedeterministického automatu má podobu $Q \times \Sigma \to 2^Q$
- s ∈ Q je počáteční stav automatu.
- F je množina koncových stavů, F ∈ Q.

Pokud je přechodová funkce definována jako $\delta : Q \times \Sigma \to Q \cup \{nedef.\}$, tzn. pokud |δ(q,a)| = 1 pro všechna $q \in Q$ a $a \in \Sigma$, potom M se nazývá deterministický konečný automat. [4]

Deterministický konečný automat M nazýváme úplný deterministický konečný automat, pokud pro všechna q∈ Q a všechna a∈ ∑ platí, že δ(q,a)∈ Q, tj. δ je totální funkcí na $Q \times \Sigma$.

Konfigurace C konečného automatu M je uspořádaná dvojice C = (q,w), kde (q,w) ∈ $Q \times \Sigma^*$.

Počáteční konfigurace je konfigurace (s, $a_1a_2a_3 \ldots a_n$)

Koncová konfigurace je konfigurace (q_F, ε), kde $q_F \in$ F, ε značí prázdný řetězec

Přechod automatu definujeme jako binární relaci $\underset{M}{\vdash}$ v množině konfigurací C

$\underset{M}{\vdash} \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*)$, která je definována takto:

$(q, aw) \underset{M}{\vdash} (q', w) \overset{def.}{\Leftrightarrow} q' \in \delta(q, a)$ pro $q, q' \in Q, a \in \Sigma, w \in \Sigma^*$ [4]

2.2 Nedeterministické automaty

Přechodovou funkci u nedeterministických automatů definujeme tak, že v každém bodě tabulky přechodů není zapsán pouze jeden následující stav, ale celá množina stavů, která je podmnožinou 2^Q (oproti deterministickému konečnému automatu, který v každém místě tabulky může obsahovat maximálně jeden cílový stav). Tyto automaty nazýváme nedeterministické konečné automaty, proto, že se při přečtení každého dalšího symbolu musí nedeterministicky rozhodnout, do jakého stavu přejdou. Za běhu pak takový automat při přečtení jednoho symbolu ze vstupu může přejít jakoby současně do všech stavů této množiny a ze všech těchto stavů pokračuje čtením dalšího vstupu. Je-li po přečtení celého vstupního řetězce automat alespoň v jednom z koncových stavů, potom je daný vstup přijat. [1]

2.3 Rozšířené konečné automaty

Rozšířený konečný automat je petice $M = (Q, \Sigma, \delta, s, F)$, kde

- Q je konečná množina všech stavů.
- Σ je konečná množina všech vstupních symbolů, nazýváme ji vstupní abeceda.
- δ je zobrazení $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$
- s ∈ Q je počáteční stav automatu.
- F je množina koncových stavů, F ∈ Q.

V přechodové tabulce rozšířeného konečného automatu je také navíc sloupeček pro prázdný vstup, označovaný ε. V celé teorii formálních jazyků existuje speciální slovo ε, které označuje prázdný řetězec, a platí, že ε ∉ Σ. V tabulce přechodů se tyto přechody nazývají epsilon-přechody. Automat tyto přechody může provádět bez načítání vstupních znaků teoreticky do nekonečna, avšak prakticky nám stačí vytvořit v každém stavu epsilon-uzávěr a místo do jednoho stavu se přesunout do všech stavů, které odpovídají tranzitivnímu uzávěru přes tyto přechody. Této vlastnosti dále využíváme při odstraňování epsilon-přechodů. [1]

Funkci, která nám pro každý stav q∈ Q spočítá epsilon-uzávěr, nazýváme ε-uzávěr(q):

ε-uzávěr(q) = $\{p \mid \exists w \in \Sigma^* : (q, w) \overset{*}{\underset{M}{\vdash}} (p, w)\}$

Funkci ε-uzávěr zobecníme tak, aby argumentem mohla být množina $T \subseteq Q$:

ε-uzávěr(T) = $\bigcup_{q \in T}$ ε-uzávěr(q)

Tyto vlastnosti rozšířených, ani vlastnosti nedeterministických automatů nepřidávají automatu větší sílu pro rozhodování vyšších tříd jazyků. Všechny konečné automaty mohou rozhodovat pouze regulární jazyky, protože jakýkoliv automat můžeme relativně jednoduchým algoritmem převést na

jiný typ automatu přijímajícího stejný jazyk. Teoreticky je velmi triviální převést libovolný nedeterministický automat na deterministický, ale prakticky je tento postup poněkud složitější. Abychom toho dosáhli, je potřeba původní množinu stavů Q nahradit její potenční množinou (množina všech podmnožin množiny Q). Každý stav takto vytvořeného automatu pak odpovídá nějaké množině stavů původního nedeterministického automatu a jsou mezi nimi jednoznačné přechody. Bohužel tímto vzroste velikost nové množiny stavů na 2^Q, dále se spousta takových stavů stane nedosažitelnými a je potom už jenom otázka výpočetních strojů a algoritmů, jak se s touto skutečností vypořádají. [1]

2.4 Popis činnosti automatu

Na počátku vždy automat začíná v předem definovaném počátečním stavu s. Ať už se jedná o jakýkoliv typ automatu, tak počáteční stav je vždy právě jeden. Při každém kroku se přečte právě jeden (nebo v případě rozšířeného automatu se nemusí přečíst žádný) symbol ze vstupního řetězce. Přečtený symbol je odstraněn ze vstupu a automat přejde do dalšího stavu na základě své přechodové tabulky σ. Takto se pokračuje se čtením vstupního řetězce, dokud není vstup prázdný, nebo dokud existuje přechodová funkce pro aktuální stav a přečtený symbol. Podle toho, zda automat skončí po přečtení vstupu ve stavu $f \in F$, platí, že automat buď daný vstup akceptuje, nebo neakceptuje. Množina všech řetězců, které je daný automat schopen přijmout, tvoří regulární jazyk. [1]

2.4.1 Ukázka činnosti

Mějme např. nedeterministický konečný automat $M = (Q, \Sigma, \delta, s, F)$

Q = { q0,q1,q2,q3 }

Σ = (0, 1)

σ: $Q \times \Sigma \rightarrow 2^Q$, viz tabulka:

stav	Symbol 0	Symbol 1
q0	q1, q2	q3
q1	q0, q3	q3
q2	q0	q0, q3
q3	q1	q0, q2

Tabulka 1 - Tabulka přechodů

s = q0

F = { q0 }

Automat bude zpracovávat řetězec 011, výpočet bude probíhat takto:

Na počátku je automat ve stavu q0, zpracovává se první načtený symbol, 0. Z tabulky přechodů si vybereme odpovídající přechod, tedy automat se bude nyní nacházet ve stavech q1 a q2.

Dalším načteným znakem bude symbol 1, nyní musíme provést přechodovou funkci pro všechny aktuální stavy, ve kterých se automat nachází. Pro stav q1 je novým stavem stav q3, pro stav q2 jsou novými stavy q0 a q3. Sjednocením všech těchto možných stavů dostaneme výslednou množinu stavů, ve které se bude automat nacházet, tedy stavy q0 a q3.

Nakonec přečteme poslední vstupní znak 1 a provedeme opět přechod ze všech aktuálních stavů. Pro stav q0 je to stav q3, pro stav q3 jsou to stavy q0 a q2. Po provedení sjednocení opět dostaneme novou množinu aktuálních stavů. Nyní se nacházíme ve stavech q0, q2 a q3.

Vyčerpali jsme všechny vstupní symboly, zároveň jsme se dostali jednou z možných cest do koncového stavu q0, takže vstupní řetězec 011 byl automatem přijat, tento řetězec patří do jazyka přijímaného tímto automatem.

2.5 Nedosažitelné stavy

Buď M = (Q, ∑, d, s, F) konečný automat. Stav q∈ Q nazýváme dosažitelný, pokud existuje w∈ ∑* takové, že $(s,w) \overset{*}{\underset{M}{\vdash}} (q,\varepsilon)$, jinak je tento stav nedosažitelný. Stav je nedosažitelný, pokud není dosažitelný ze startovního stavu s. [2]

Algoritmus, který eliminuje nedosažitelné stavy, postupuje právě ze startovního stavu a postupně prochází celý automat do šířky, dokud neprojde všechny dosažitelné stavy, nedosažitelné stavy se poté mohou ignorovat, jelikož i bez nich bude automat přijímat ekvivalentní regulární jazyk.

Algoritmus eliminace nedosažitelných stavů:

Vstup: Deterministický konečný automat $M = (Q, \Sigma, \delta, s, F)$

Výstup: Deterministický konečný automat M´ bez nedosažitelných stavů, L(M) = L(M´)

Metoda:

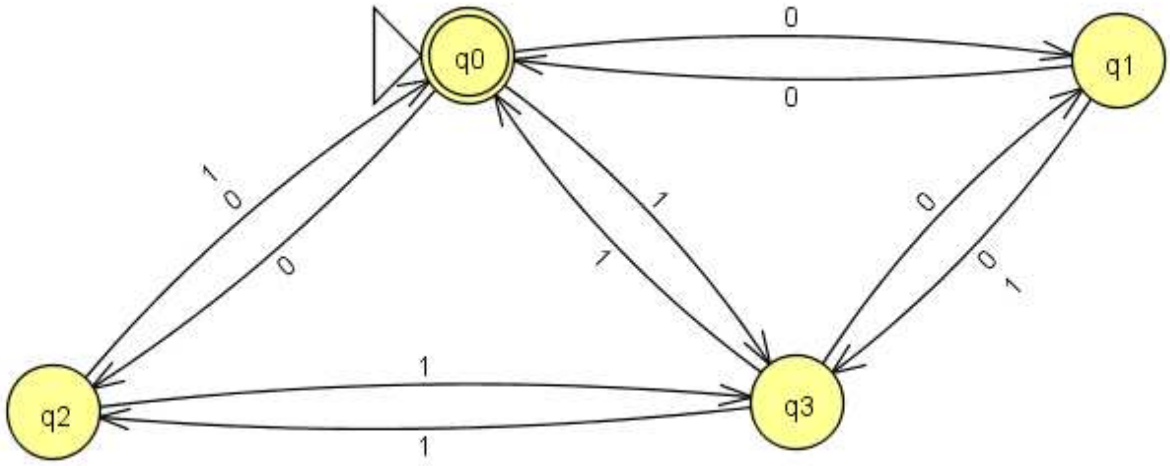
1. i := 0
2. S_i := {q0}
3. repeat
4. $S_{i+1} := S_i \cup \{q \mid \exists p \in S_i \exists a \in \Sigma : \delta(p,a) = q\}$
5. i := i + 1
6. until $S_i = S_{i-1}$
7. $M' := (S_i, \Sigma, \delta_{S_i}, s, F \cap S_i)$ [3]

2.6 Grafické znázornění konečného automatu

Popis přechodové funkce pomocí tabulky může být zvláště pro velké automaty velice nepřehledné. Místo této tabulky se pro popis konečného automatu používá grafické znázornění všech stavů a

přechodů mezi nimi. Kolečky znázorňujeme jednotlivé stavy. Pomocí šipek znázorňujeme přechody. Pokud existuje přechod $q2 \in \delta(q1,a)$, $q1,q2 \in Q$, $a \in \Sigma$, potom vede šipka ze stavu q1 do stavu q2, a je označena symbolem a.

Příklad takového znázornění pro předchozí ukázkový automat je na obrázku 1.

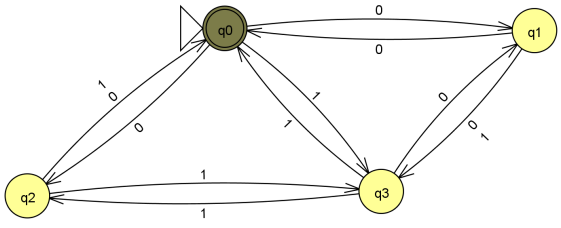


Obrázek 1 - Grafické znázornění konečného automatu

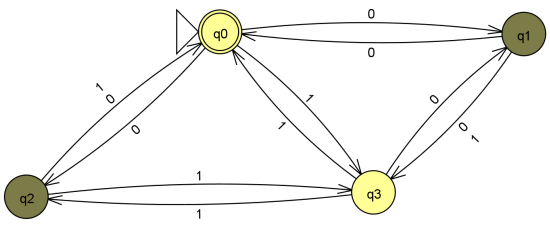
Dvojité kolečko označuje koncové stavy (zde je to pouze jeden, q0), počáteční stav je označen šipkou, někdy s připsaným textem, např. START. (Tato notace není jediná, jindy se např. koncové stavy označují tlustším orámováním a dvojité kolečko označuje počáteční stav apod.) já však budu ve své práci používat tuto notaci.

Nyní si můžeme ještě znázornit, jak bude vypadat přijetí řetězce 011 z minulé kapitoly.

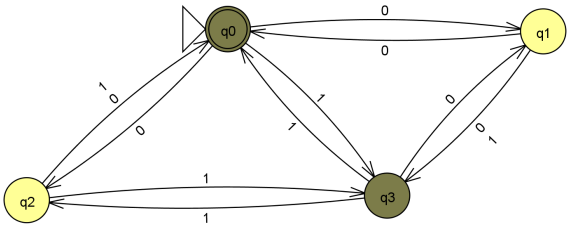
Obrázek 2 - Počáteční konfigurace automatu



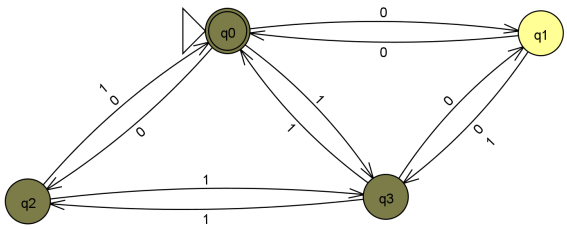
Obrázek 3 - Načten symbol 0



Obrázek 4 - Načten symbol 1



Obrázek 5 - Načten symbol 1



Na obrázcích 2 – 5 je graficky znázorněn běh automatu M. V každém kroku jsou znázorněny stavy, ve kterých se může aktuálně nacházet. Po přečtení posledního znaku (obr. 5) je mnohem lépe vidět, že automat se mimo jiné nachází v koncovém stavu q1, a tudíž přijímá zadaný řetězec 011.

2.7 Determinizace konečného automatu

Pro převod nedeterministického automatu na deterministický existuje následující formální algoritmus:

Vstup: Nedeterministický konečný automat $M = (Q, \Sigma, \delta, q_0, F)$

Výstup: Deterministický konečný automat $M' = (Q', \Sigma, \delta', q_0', F')$

Metoda:

1. Polož $Q' = (2^Q \setminus \{\emptyset\}) \cup \{nedef.\}$

2. Polož $q_0' = \{q_0\}$

3. Pro všechna $S \in 2^Q \setminus \{\emptyset\}$ a pro všechna $a \in \Sigma$ polož: $\delta'(S,a) = \bigcup_{q \in S} \delta(q,a)$

Je-li $\delta'(S,a) = \emptyset$, polož $\delta'(S,a)$ = *nedef.*

4. Polož $F' = \{S \mid S \in 2^Q \wedge S \cap F \neq \emptyset\}$

Po skončení tohoto algoritmu bývá zpravidla většina stavů z množiny Q' nedostupných. K nalezení funkce δ' aplikujeme zkrácený postup využívající tuto skutečnost. Taktéž můžeme začít s počátečním stavem q_0' a pro všechny symboly vstupní abecedy určit následný stav, takto můžeme pokračovat pro všechny nově vytvořené stavy, dokud budou vznikat. Nakonec dostaneme deterministický konečný automat bez nedostupných stavů, protože všechny jeho stavy jsme vlastně vytvořili průchodem automatu ze startovního stavu. Princip determinizace automatu spočívá tedy v prohledání do šířky (breadth-first-search) a nalezení všech možných stavů, do kterých se může automat ze startovního stavu dostat.

Nyní determinizujeme předchozí automat z obr. 1. Začneme v počátečním stavu a pro každý symbol z abecedy vytvoříme nový stav.

Stav	Symbol 0	Symbol 1
q0	q1, q2	q3

Tabulka 2 - První krok determinizace

Nyní nám vznikly dva nové stavy, {q1,q2} a {q3}. Pro stav {q1,q2} platí, že následující stav bude zahrnovat všechny stavy, do kterých se můžeme dostat jak ze stavu q1, tak i ze stavu q2.

Stav	Symbol 0	Symbol 1
q1,q2	q0,q3	q0,q3
q3	q1	q0,q2

Tabulka 3 - Druhý krok determinizace

Tentokrát nám vznikly další tři nové stavy, pokračujeme dál, dokud budou vznikat.

Stav	Symbol 0	Symbol 1
q0,q3	q1,q2	**q0,q2,q3**
q1	q0,q3	q3
q0,q2	**q0,q1,q2**	q0,q3
q0,q2,q3	q0,q1,q2	q0,q2,q3
q0,q1,q2	**q0,q1,q2,q3**	q0,q3
q0,q1,q2,q3	q0,q1,q2,q3	q0,q2,q3

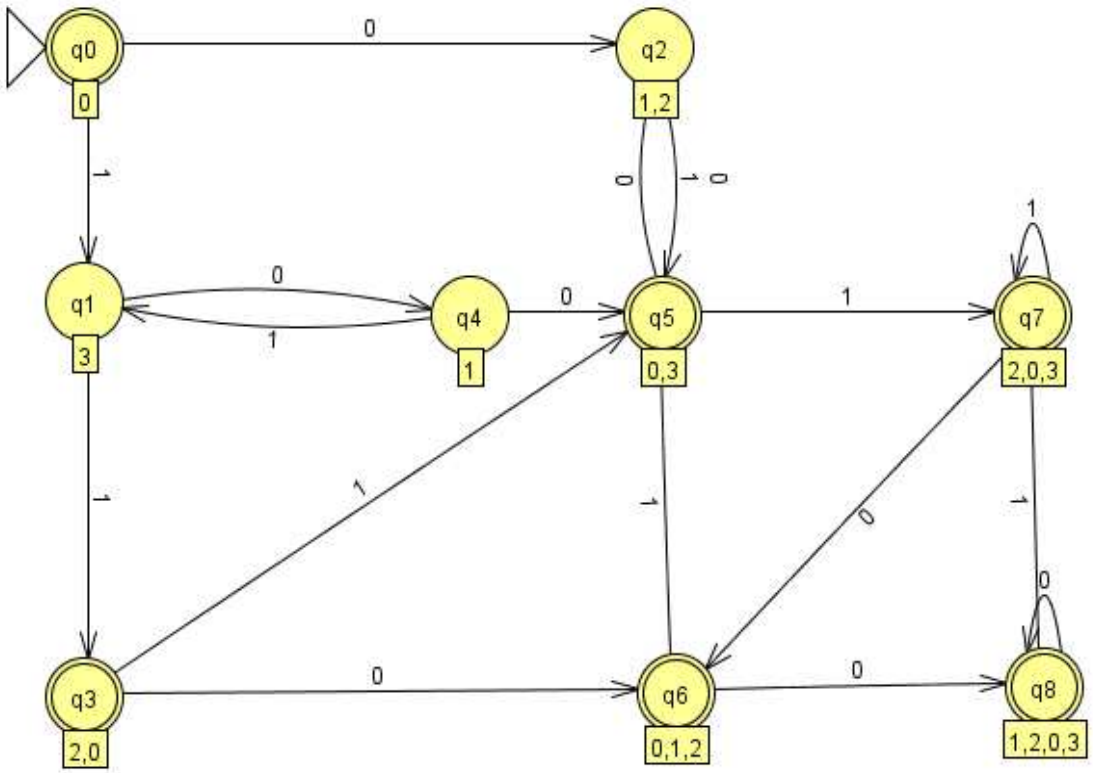
Tabulka 4 - Třetí krok determinizace

Zvýrazněné jsou ty stavy, které v automatu přibyly jako nové a jsou v tabulce dále zpracovány. Takto jsme získali tabulku přechodové funkce pro nový automat M´

Stav	Symbol 0	Symbol 1
q0	q1, q2	q3
q1,q2	q0,q3	q0,q3
q3	q1	q0,q2
q0,q3	q1,q2	q0,q2,q3
q1	q0,q3	q3
q0,q2	q0,q1,q2	q0,q3
q0,q2,q3	q0,q1,q2	q0,q2,q3
q0,q1,q2	q0,q1,q2,q3	q0,q3
q0,q1,q2,q3	q0,q1,q2,q3	q0,q2,q3

Tabulka 5 - Poslední krok determinizace

Automat M´ bude mít všech 9 stavů z tabulky. Nepotřebovali jsme vytvořit všech 2^Q stavů, protože všechny ostatní stavy by byly nedosažitelné. Tímto algoritmem jsme vytvořili deterministický automat, který nemá žádné nedosažitelné stavy. Výsledný automat bude vypadat následovně:



Obrázek 6 - Deterministický konečný automat M´

2.8 Minimalizace konečného automatu

Minimalizovat (neboli redukovat počet stavů na minimum) můžeme pouze deterministické konečné automaty. Před tím, než budeme moci minimalizovat rozšířené konečné automaty, budeme z nich muset nejdříve odstranit epsilon-přechody a převést na deterministický konečný automat. Samotný převod deterministického konečného automatu na minimální konečný automat se potom skládá z několika kroků, nejdříve eliminujeme nedosažitelné stavy, zredukujeme nerozlišitelné stavy, převedeme automat na redukovaný deterministický konečný automat a odstraníme přebytečné stavy, které nemají vliv na přijímání vstupního řetězce. Postup pro eliminaci nedosažitelných stavů je v kapitole 2.5.

Základem algoritmu minimalizace počtu stavů je koncept nerozlišitelných stavů. [3]
Nechť $M = (Q, \Sigma, \delta, s, F)$ je úplný deterministický konečný automat.

Říkáme, že řetězec w∈ ∑* rozlišuje q1,q2, jestliže $(q1, w) \overset{*}{\underset{M}{\vdash}} (q3, \varepsilon) \wedge (q2, w) \overset{*}{\underset{M}{\vdash}} (q4, \varepsilon)$ pro nějaké q3, q4 a právě jeden ze stavů q3, q4 je v F.

Dále říkáme, že stavy q1, q2 ∈ Q jsou k-nerozlišitelné (zapisujeme takto: $q1 \overset{k}{\equiv} q2$), právě když neexistuje žádný řetězec w∈ ∑*, |w| ≤ k, který rozlišuje q1, q2.
Stavy q1 a q2 jsou nerozlišitelné, pokud pro každé k ≥ 0 jsou k-nerozlišitelné, zapisujeme $q1 \equiv q2$

Deterministický konečný automat M nazýváme redukovaný, jestliže žádný stav q∈ Q není nedostupný a zároveň žádné dva stavy q1,q2∈ Q nejsou nerozlišitelné. [5]

Algoritmus převodu na redukovaný deterministický automat:

Vstup: Úplný deterministický konečný automat $M = (Q, \Sigma, \delta, s, F)$

Výstup: Redukovaný deterministický konečný automat $M' = (Q', \Sigma, \delta', s', F')$, L(M) = L(M´)

Metoda:

1. Odstranění nedostupných stavů
2. i := 0
3. $\overset{0}{\equiv} := \{(p,q) \mid p \in F \Leftrightarrow q \in F\}$
4. repeat
5. $\overset{i+1}{\equiv} := \left\{(p,q) \mid p \overset{i}{\equiv} q \wedge \forall a \in \Sigma : \delta(q,a)\right\}$
6. i := i + 1
7. until $\overset{i}{\equiv} = \overset{i-1}{\equiv}$
8. $Q' := Q / \overset{i}{\equiv}$
9. $\forall p,q \in Q \forall a \in \Sigma : \delta'([p],a) = [q] \Leftrightarrow \delta(p,a) = q$
10. s´ = [s]
11. F´ = { [q] | q∈ F }

Výraz [x] značí ekvivalenční třídu určenou prvkem x.

Dále budeme minimalizovat náš determinizovaný automat M (viz obr. 6). První krok již máme za sebou, protože automat neobsahuje žádné nedostupné stavy. Vytvoříme první ekvivalenční třídu rozdělenou na dvě skupiny, v první skupině budou všechny koncové stavy (q0,q3,q5,q6,q7,q8), v druhé skupině budou nekoncové stavy (q1,q2,q4). Nyní budeme postupovat v algoritmu v bodě 4 až 6, dokud nebudou ve všech skupinách pouze jazykově nerozlišitelné stavy. Postup algoritmu je možné znázornit tabulkami. V každé tabulce jsou vždy stavy odděleny do skupin, kde v každé skupině jsou stavy, které jsou k-nerozlišitelné.

	$\overset{0}{\equiv}$	Symbol 0	Symbol 1
I	q0	q2 II	q1 II
I	q3	q6 I	q5 I
I	q5	q2 II	q7 I
I	q6	q8 I	q5 I
I	q7	q8 I	q7 I
I	q8	q8 I	q7 I
II	q1	q4 II	q3 I
II	q2	q5 II	q5 II
II	q4	q5 II	q1 II

Po prvním kroku vidíme, že skupina číslo I se bude nyní rozpadat na tři skupiny a skupina II se nyní rozpadne na dvě skupiny. Celkem tedy budeme v další iteraci uvažovat pět skupin, jejichž stavy jsou prozatím 1-nerozlišitelné. Nově vzniklé skupiny budou:

I: q0

II: q5

III: q3,q6,q7,q8

IV: q1

V: q2,q4

	1 ≡	Symbol 0	Symbol 1
I	q0	q2 V	q1 II
II	q5	q2 II	q7 III
III	q3	q6 III	q5 II
III	q6	q8 III	q5 II
III	q7	q8 III	q7 III
III	q8	q8 I	q7 I
IV	q1	q4 V	q3 III
V	q2	q5 II	q5 II
V	q4	q5 II	q1 V

Ve druhém kroku dochází opět k rozkladu některých ze skupin. Všimněme si, že skupiny číslo I, II a IV obsahují pouze jeden prvek, to znamená, že se již dále nerozpadají, proto bychom pro ně již nemuseli vypočítávat jejich hodnoty a zapisovat je do tabulky. Jak dále vidíme, tak i skupina číslo V se celá rozpadla. Nyní nám ještě zbývá poslední skupina, která má ještě více než jeden prvek., to jsou stavy q3 a q6. Nyní nám stačí pouze vypočítat hodnoty pro tuto skupinu.

	2 ≡	Symbol 0	Symbol 1
I	q3	q6	q5
I	q6	q8	q5

I tato poslední skupina se nakonec rozpadne, takže dále už nebudeme muset počítat a můžeme prohlásit, že daný automat byl již na vstupu redukovaný, že aplikováním redukčního algoritmu jsme dostali úplně stejný automat, jaký byl na začátku.

3 Paralelní programování

Paralelní programování je způsob, jakým lze psát programy tak, aby bylo možno provádět co nejvíce operací současně, a využít tak optimálně celý výpočetní stroj. Velké problémy lze velmi často rozdělit na menší, které je možno řešit současně. Existuje několik různých forem paralelních výpočtů. Můžeme je nalézt např. na bitové úrovni, na úrovni instrukcí, dat, nebo celých úkolů. [7]

Paralelní programy mohou být spuštěny na několika fyzických procesorech, proto musí být zpracovávaný problém rozdělen na podproblémy, které je možno řešit současně, tzn. že, nesmí být mezi nimi datová závislost. Žádný program nemůže běžet rychleji než jeho nejdelší datově závislá část, protože tato část programu musí vždy proběhnout v daném pořadí. Naštěstí spousta algoritmů neobsahují dlouhé závislé bloky, takže v nich můžeme hledat paralelní části, které mohou být vypočítány současně. [7]

Paralelní zpracování je vhodné především ve vysoce náročných výpočtech. Zájem o paralelní programování v dnešní době roste v důsledku fyzického omezení frekvence procesorů. Zvyšování frekvence procesorů bylo možné od osmdesátých let dvacátého století do roku 2004, dokud to bylo pořád fyzicky možné. Čas zpracování programu na jednom procesoru byla rovna počtu instrukcí násobena průměrnou dobou zpracování jedné instrukce. Zvětšení frekvence procesoru nebo snížení doby průměrné instrukce přinášelo zrychlení. [7]

3.1 Flynnova klasifikace

Michael J. Flynn vytvořil jeden z prvních klasifikačních systémů pro paralelní i sekvenční počítače a programy, známý jako Flynnova klasifikace. Flynn rozdělil počítače a programy podle toho, zda zpracovávají jednu nebo více instrukcí naráz a dále zda instrukce používají jedna nebo více dat. [6]

	Single instruction	Multiple instruction
Sdngle data	SISD	MISD
Multipl e data	SIMD	MIMD

SISD je ekvivalentní sekvenčnímu programu bez náznaku paralelismu. SIMD zpracovává opakovaně stejnou operaci na velké množině dat. To se běžně provádí v aplikacích pro zpracování signálů. MISD je zřídka používaná klasifikace. Více operací dostane na vstup stejná data. Programy MIMD jsou nejčastějším typem paralelních programů. Počítače podporující MIMD mají několik procesorů, které fungují asynchronně a nezávisle. V každém okamžiku může každý procesor provádět jinou instrukci na jiné části dat. [6]

3.2 Modely paralelního programování

Existuje několik modelů paralelního programování: Sdílená paměť, Vlákna, Předávání zpráv, Paralelní model dat, hybridní model. [6]

3.2.1 Sdílená paměť

V tomto modelu se sdílenou pamětí, všechny procesy sdílí jeden společný adresový prostor v paměti, ze kterého mohou číst i zapisovat asynchronně. Proto musí být v tomto případě implementovány zámky nebo semafory pro přístup k jednotlivým položkám v paměti. Výhodou tohoto modelu je z programátorského hlediska, že žádná proměnná nemá specifického vlastníka, každý proces může používat celý paměťový prostor, takže není potřeba explicitně specifikovat komunikaci mezi procesy. Program lze takto často zjednodušit. [6]

Na platformách se sdílenou pamětí musí být překladače schopny přeložit proměnné programu na skutečné adresy, které jsou globální.

3.2.2 Vlákna

V tomto modelu paralelního programování může mít jeden proces více běžících částí, každá se stará o svoji část programu. Každé vlákno takového programu spouští svoji funkci, kterou provádí až do konce. [6]

Hlavní program je naplánován a spuštěn operačním systémem. Naalokují se všechny potřebné systémové a paměťové zdroje. Hlavní program provede inicializaci a poté vytvoří potřebný počet vláken, která budou naplánována a řízena operačním systémem, tato vlákna bude systém podle možností spouštět asynchronně, nezávisle na sobě. Každé vlákno programu má svoje vlastní lokální data ale také sdílí všechny zdroje hlavního procesu. To nám šetří režii spojenou s replikací všech zdrojů pro každé vlákno. Práce jednotlivých vláken je napsána ve funkcích, přičemž více vláken může naráz spouštět tu samou funkci, a mohou provádět stejnou činnost např. s jinými daty.

Vlákna mezi sebou komunikují pomocí globální paměti, takže je potřeba synchronizace, aby nedošlo k tomu, že by dvě vlákna zapisovala ve stejný čas do jednoho paměťového místa. [6]

Vlákna mohou libovolně vznikat a po dokončení své práce i zanikat, vše musí hlídat hlavní program. Hlavní program běží po celou dobu všech výpočtů, aby měla vlákna přístup ke všem sdíleným prostředkům. [6]

Z pohledu programátora, implementace vláken obvykle zahrnují knihovny funkcí, které jsou volány ze vnitř paralelního zdrojového kódu, nebo sadu direktiv překladače zapsaných přímo do sériového nebo i paralelního kódu. V obou případech je však potřeba, aby programátor určil všechny paralelismy. [6]

3.2.3 Předávání zpráv

Různé úkoly nemusí během výpočtu pracovat se sdílenou pamětí. Více úkolů může být umístěno na stejném fyzickém stroji, stejně tak i na více počítačích. Při rozmístění úkolů na různé výpočetní stroje můžeme dosáhnout několikanásobného výkonu, stejně tak jako kdybychom tyto úkoly počítali na víceprocesorovém stroji. Zde je důležité zasílání zpráv mezi jednotlivými počítači, je potřeba, aby si mezi sebou předávaly úkoly nebo již vypočítané výsledky. Přenos dat musí být podporována na všech strojích a všechny stroje musí používat stejný komunikační protokol. V těchto programech je navíc potřeba nějaká komunikační knihovna, která bude umožňovat zasílání zpráv po síti. Tyto knihovny vznikaly od osmdesátých let dvacátého století, ale jejich implementace se většinou lišily natolik, že nebylo dost dobře možné vytvářet dobré přenositelné programy. V roce 1992 přišlo MPI Forum s cílem vytvořit standardní rozhraní pro tyto knihovny. MPI je dnes už standardem pro předávání zpráv. Na architekturách se sdílenou pamětí nepoužívají MPI implementace pro komunikaci počítačovou síť, místo toho používají sdílenou paměť. [6]

3.2.4 Paralelní model dat

Existuje spousta úloh, které provádí tutéž operaci nad různými daty v nějaké množině. Touto množinou je typicky nějaké pole hodnot, matice, nebo kostka, výsledky jednotlivých hodnot na sobě nejsou nijak závislé, proto lze tyto výpočty počítat velmi efektivně pomocí paralelního zpracování. Množina všech hodnot je rozdělena na patřičný počet částí, které je možné na jednom fyzickém stroji zpracovat současně. Nad každou tuto část může být provedena požadovaná operace paralelně. Např. přičtení konstanty ke všem prvkům pole. [6]

Na architektuře se sdílenou pamětí, všechny prováděné operace mají přístup k celé datové struktuře pomocí globální paměti, každá operace si vybírá pomocí indexů pouze ty hodnoty, které sama zpracovává. Na architektuře s distribuovanou pamětí je množina rozdělena na kousky a každý proces má ve své lokální paměti uloženou svoji část. [6]

3.2.5 Hybridní model dat

V tomto modelu jsou zkombinovány dva nebo více paralelních programovacích modelů dat. V současné době je typickým příkladem model předávání zpráv (MPI) kombinovaný buď s vláknovým modelem (POSIX) nebo s modelem sdílené paměti (OpenMP). Tento hybridní model se hodí i do prostředí počítačových sítí. [6]

Ještě jedním běžným příkladem je hybridní model kombinující paralelní model dat a předávání zpráv. [6]

3.3 OpenMP

Jedním z nejznámějších aplikačních rozhraní je OpenMP API. Podporuje programovací jazyky C/C++ a Fortran. Je to multiplatformní knihovna pro počítače se sdílenou pamětí jak pro operační systémy Windows, tak i pro Linux a obsahuje direktivy pro překladač pro snadnou implementaci paralelních programů. Tato knihovna byla vytvořena skupinou hlavních výrobců softwaru a hardwaru. OpenMP je přenosný škálovatelný model, který nabízí programátorům při psaní paralelních programů jednoduché a flexibilní rozhraní pro usnadnění vývoje těchto aplikací, ať už na běžných domácích počítačích, nebo na superpočítačích. Je to volně šířitelná knihovna dostupná na internetu. [9]

Pouze programátor je při psaní kódu zodpovědný za vyhledání a označení paralelních částí. Tím dostává programátor volnou ruku, protože může chtít např. paralelně vykonat pouze nějaké kritické části kódu. Pro správný překlad do paralelního programu je zapotřebí mít nainstalováno OpenMP API rozhraní a preprocesor pro daný programovací jazyk. Avšak pokud není k dispozici na cílovém počítači, dá se pořád ještě kód přeložit do neparalelního programu. [10]

V OpenMP jsou tři hlavní části, první a nejdůležitější částí jsou direktivy pro preprocesor, dále jsou to vestavěné knihovní funkce, které ale pro práci není nutné používat, a na konec jsou to systémové proměnné, které uchovávají stav aplikace, např. OMP_NUM_THREADS. [10]

Příkladem může být jednoduchá ukázka v jazyce C, která vytvoří několik vláken a každé z nich vypíše na výstup svou hodnotu.

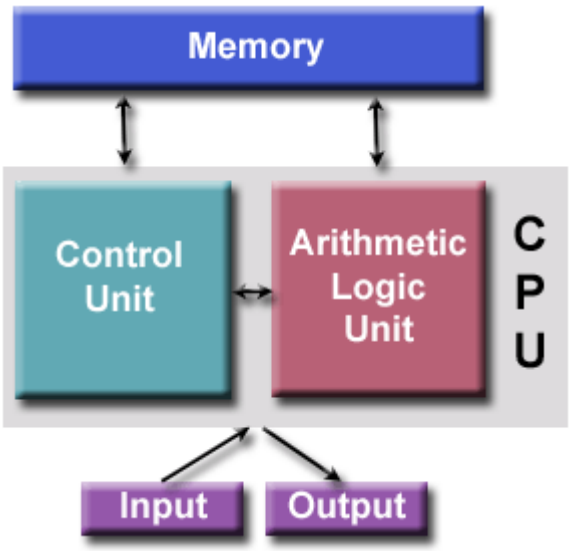
```
void main() {
int i, j;
i = 10;
#pragma omp parallel
{
    j = 20;
   printf("%i\n", j);
};
printf("Ahoj\n");
}
```

Pomocí direktivy parallel v tomto kódu říkáme, že blok kódu, který bezprostředně následuje na dalším řádku, se provede paralelně, každé vytvořené vlákno zpracuje stejnou část, stejný kód. Zde všechna vlákna přiřadí hodnotu do proměnné j a následně ji vytiskne na výstup. Po skončení všech vláken bude pokračovat hlavní program dál, a teprve poté se jen jednou vypíše „Ahoj“.

OpenMP se vždy stará o správné vytvoření vláken i o jejich úklid, proto je paralelní programování pomocí tohoto nástroje velmi efektivní z hlediska psaní zdrojových kódů. Spousta sekvenčních programů by se takto jednoduše dala urychlit a optimalizovat pro spuštění na vícejádrových strojích

4 Hardwarová architektura

Hardwarová architektura má veliký vliv na běh programů, nejen paralelních, ale i sériových. Odvíjí se od ní rychlost zpracování jednotlivých příkazů. Celý počítač se skládá z několika nezávislých částí, které spolu kooperují a předávají si mezi sebou data. Prvním člověkem, který navrhl strukturu počítače, byl von Neumann, jehož myšlenka se stala základem pro dnes běžné architektury.



Již v roce 1945 popsal první obecné požadavky na elektronické počítače. Od této doby prakticky všechny počítače začaly používat tento model. Skládá se ze 4 hlavních částí: Paměť, řídící jednotka, aritmeticko-logická jednotka a vstupní/výstupní jednotka. Paměť umožňuje čtení i zápis a slouží k uložení programu i dat. Program je sada instrukcí, které počítač vykonává, data jsou pouze informace, které program používá k činnosti. [6]

Obrázek 7 - Von Neumannova architektura

Řídící jednotka přenáší instrukce/data z paměti, dekóduje instrukce a sekvenčně je provádí tak, aby mohl být naplánovaný úkol zpracován. Aritmeticko-logická jednotka slouží k provádění základních aritmetických operací. Vstupní/Výstupní jednotka slouží ke komunikaci s lidmi. [6]

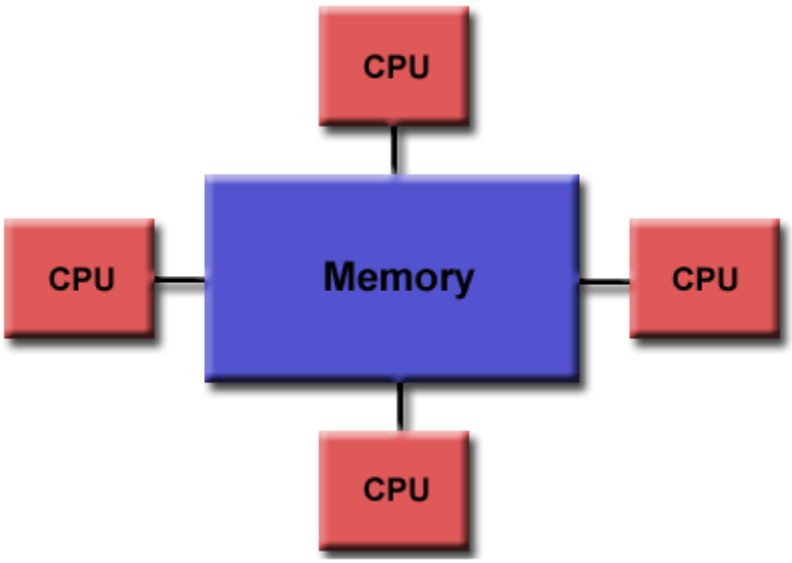
Rychlost počítače se většinou udává ve frekvenci, dnes jsou to řádově GHz. To udává počet cyklů procesoru za jednu sekundu. Nicméně, toto srovnání je poněkud zavádějící, protože stroj s vyšší frekvence nemusí nutně mít vyšší výkon. Výkon počítače lze také ovlivnit velikostí paměti CACHE, proto čím větší má procesor frekvenci a čím větší má vyrovnávací paměť, tím je výkonnější. Moderní procesory mohou vykonávat více instrukcí během jednoho cyklu, což výrazně urychluje běh programu. Dalšími faktory ovlivňujícími rychlost programu jsou rychlost sběrnice, volná paměť, typ a pořadí instrukcí. Dále rozlišujeme dva pojmy, zpoždění a propustnost. Zpoždění je doba mezi začátkem procesu a dokončením. Propustnost je množství práce vykonaná za jednotku času. [12] Důležitou součástí každého počítače jsou paměti. Jsou hierarchicky uspořádané, aby se dosáhlo optimálního využití rychlosti. Nejbližší paměťové buňky jsou přímo v procesoru a říká se jim registry. Jsou to nejrychlejší paměťová místa v počítači, je jich však velmi omezený počet, řádově stovky registrů. Dále je na procesoru paměť cache, která je stále velmi rychlá, avšak je velice drahá a její velikost se pohybuje v řádu MB. Pomalejší je potom hlavní paměť počítače (RAM) a další. [12]

Ve všech těchto hlavních skupinách se používá mnoho různých typů pamětí, které se zásadně liší svými parametry, fyzikálními principy, způsobem výběru a dalšími vlastnostmi. Výkonnost je u pamětí udána parametry: kapacita, přístupová doba a přenosová rychlost. Přístupová doba je doba od zahájení čtení po získání obsahu paměti ze zvoleného místa. Doba cyklu je doba od zahájení čtení nebo zápisu až po skončení této operace, kdy je poté dále možno spustit další operaci čtení/zápis. Přenosová rychlost je parametr udávající počet datových jednotek (bitů, bytů atd.) přenášených z nebo do paměti za sekundu, např. 5MB/s u disku. [11]

4.1 Architektury pamětí paralelních počítačů

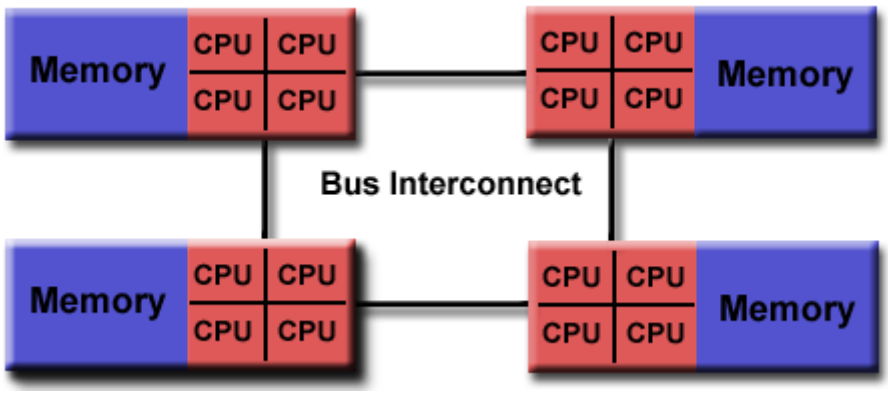
4.1.1 Sdílená paměť

Paralelní počítačové architektury se sdílenou pamětí mají pouze jednu společnou paměť, ke které mohou přistupovat globálně všechny procesory. Všechny procesory pracují nezávisle na sobě, ale používají jeden sdílený adresový prostor. Změny, které v paměti provede jeden procesor, jsou okamžitě přístupné dalším procesorům. Stroje se sdílenou pamětí lze rozdělit do dvou hlavních kategorií na základě doby přístupu do paměti: UMA (obr. 8) a NUMA (obr. 9). [6]



Obrázek 8 - UMA

Dnes nejčastěji zastoupené multiprocesorové symetrické stroje používají uniformní přístup do paměti. Všechny procesory jsou identické. Časy přístupu do paměti jsou pro všechny procesory stejné. Je zde koherentní implementace pamětí cache, takže pokud jeden procesor zapíše do sdílené paměti, tak všechny procesory vědí o provedené změně. Této koherence je dosaženo na hardwarové úrovni. [6]

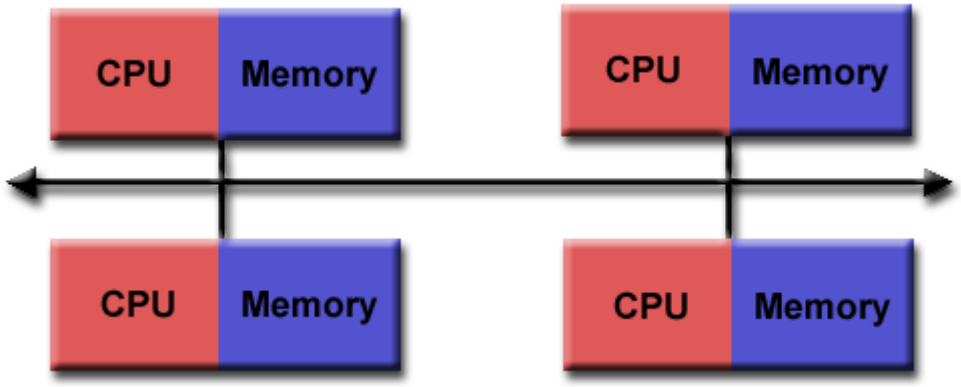


Obrázek 9 – NUMA

NUMA je často vytvořena pomocí dvou nebo více symetrických multiprocesorů. Každý má vlastní část paměti. Všechny multiprocesory mohou přistupovat do všech částí pamětí. Časy přístupů do všech částí pamětí nejsou stejné, nejrychlejší je přístup do vlastní části paměti, přístup do paměti ostatních multiprocesorů je o něco pomalejší, musí být k tomu použita sběrnice. I v případě NUMA je možné implementovat koherentní cache na hardwarové úrovni. [6]

Výhodou architektury se sdílenou pamětí je především v uživatelském přístupu. Globální adresový prostor poskytuje uživatelsky přívětivou perspektivu v programování. Sdílení dat mezi úkoly je rychlé a jednotné. Primární nevýhodou je nedostatek škálovatelnosti mezi procesory a pamětí. Přidáním více procesorů se geometricky zvyšuje provoz na sdílené paměti. Návrh a výroba strojů s větším počtem procesorů je v dnešní době pořád nákladnější. Dále je programátor zodpovědný za synchronizaci procesů, která musí být implementována pro správný přístup do globální paměti. [6]

4.1.2 Distribuovaná paměť



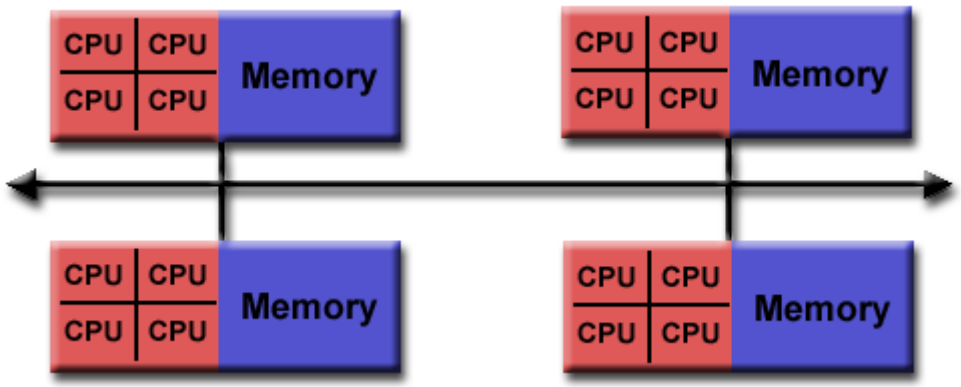
Obrázek 10 - Distribuovaná paměť mezi procesory

Systémy s distribuovanou pamětí vyžadují komunikační síť pro propojení všech pamětí mezi procesory. Každý procesor má svoji lokální paměť. Paměťová adresa v jednom procesoru se

namapuje na druhý procesor, takže zde není žádný koncept globální paměti pro všechny procesory. Protože každý procesor má svoji vlastní paměť, mohou pracovat nezávisle. Změny prováděné v lokální paměti nemají žádný efekt na paměti jiných procesorů. Pojem koherence cache zde neplatí. Pokud potřebuje procesor data z paměti jiného procesoru, pak je úkolem programátora explicitně definovat, jakým způsobem si procesory budou data mezi sebou předávat. Synchronizace procesů je rovněž práce programátora. [6]

Výhodou je škálovatelnost počtu procesorů. Velikost paměti a počtu procesorů se zvyšuje úměrně. Každý procesor může velmi rychle přistoupit k vlastní paměti bez nutnosti starat se o cache koherenci. Na druhou stranu je opět programátor zodpovědný za meziprocesorovou komunikaci a předávání dat mezi procesory. Časový přístup ke všem datům je opět neuniformní. [6]

4.1.3 Hybridní architektura



Obrázek 11 - Hybridní architektura pamětí

Největší a nejrychlejší počítače na světě dnes kombinují obě architektury, sdílenou i distribuovanou paměť. Jedna sdílená paměťová komponenta u sebe má několik sériových multiprocesorů a je zde implementována cache koherence. Každý tento multiprocesor může k danému úseku paměti přistupovat jako ke globálnímu úložišti. Všechny tyto komponenty jsou spojeny komunikační sítí jako v distribuované paměťové architektuře. Všechny multiprocesory v jedné komponentě vědí pouze o vlastní sdílené paměti. Současné trendy naznačují, že tento typ paměťové architektury bude i nadále převládat na vysoce výkonných počítačích. Výhody i nevýhody jsou společné jako u obou předchozích architektur. [6]

5 Profilovací nástroje

Při psaní náročných programů může být užitečné použít tzv. profilovaní nástroje. Profilování kódu nám umožňuje zjistit, ve kterých částech program strávil nejvíce svého času a které funkce byly volány nejčastěji. Tyto informace můžeme analyzovat a zjistit, které části našeho programu jsou pomalejší, než jsme očekávali. Zjistíme také, které funkce jsou volány častěji a které méně častěji. Tím můžeme odhalit i chyby, kterých bychom si jinak nemuseli všimnout.

Vzhledem k tomu, že profilování se provádí za běhu vlastního testovaného programu, lze je použít na programy, které jsou velké nebo příliš složité na to, abychom analyzovali pouze zdrojový kód a hledali v něm závislosti. Samozřejmě profilování také závisí vždy na konkrétním běhu programu, každý program může pokaždé běže trochu jinak vzhledem ke svým parametrům. Takže jedna profilová analýza se vztahuje vždy pouze k jednomu běhu programu. To nám ovšem nemusí vadit, pokud budeme program profilovat vícekrát, nebo pokud použijeme nejčastěji používanou konfiguraci běhu programu. K profilování si můžeme vybrat jen ty funkce, které chceme analyzovat. Přesněji řečeno si můžeme zvolit, které funkce při profilování nebudou analyzovány. Žádné profilovací informace o této funkci nebudou zaznamenány.
Profilování má několik kroků:

1. Program musíme přeložit s povolením pro profilování
2. Spustíme program s danými parametry a vytvoříme datový soubor profilu.
3. Vytvořený datový soubor analyzujeme a zjistíme závislosti jednotlivých funkcí.

Samotné profilování se obvykle skládá z posledních dvou kroků. Prvním krokem je pouze překlad programu. Druhý krok nazýváme též měření. Zaznamenává se profil běžícího programu a tím získáváme statistické údaje o běžícím programu. Tyto údaje se zapisují do pomocného losovacího souboru. Jsou zde ukládány především počty volání jednotlivých funkcí, počty iterací v jednotlivých smyčkách, počty přístupů do paměti (RAM i CACHE), je možné zaznamenávat i počty přístupů na disk nebo k jiným médiím. K těmto údajům se navíc ještě ukládá doba strávená při těchto činnostech.

Třetí krok profilování nazýváme profilovací analýza. V tomto kroku analyzujeme předchozí losovací soubor a získáváme z naměřených údajů různé statistiky. K tomu můžeme použít různé existující programy s grafickým uživatelským rozhraním, které umožňují nejenom spočítat, která funkce byla volána nejčastěji, ale dokáží se na naměřená data podívat z různých úhlů a zobrazit výsledky přehledně pomocí tabulek nebo grafů.

Pomocí profilovací analýzy se obvykle snažíme vytipovat ta místa programu, v nichž program tráví nejvíce času. Takto získáme přehled o tom, jak se jednotlivé funkce podílejí na celkové době běhu programu nebo na přístupu k jiným. Na základě těchto zjištění se můžeme lépe rozhodnout, která místa a jakým způsobem budeme optimalizovat.

5.1 Profilování

Samotné profilování se skládá tedy ze dvou kroků, měření a analýzy. Profilovací programy se proto rozdělují na dvě skupiny. První skupina nástrojů se zaměřuje na sbírání údajů o běhu programu, další programy se pak zaměřují na analýzu a grafickou prezentaci naměřených dat. [13]
Sběr údajů o běhu programu úzce souvisí s instrumentací programů. Sami ji občas nevědomky používáme, když do svého kódu doplníme pomocné výpisy, pokud ladíme program a hledáme nějakou chybu. Jednoduše si můžeme vytvořit vlastní analýzu tak, že do funkce přidáme výpisy. Pomocí nichž můžeme poté zjistit, kolikrát program navštívil sledovaná místa v programu. Pro tento případ je vhodné umístit výpisy na začátky a konce všech funkcí a také všech smyček, protože to jsou kritická místa, kde program typicky tráví nejvíce svého času. Ovšem takovýto přístup lze využít prakticky velice ojediněle, protože je to velice pracné a nespolehlivé. Takto bychom mohli sledovat pouze některá místa programu, snadno bychom mohli zapomenout na nějaká významná místa v programu. Nespornou nevýhodou takového přístupu je fakt, že náš zdrojový kód by se stával čím dál více nepřehledným a mohli bychom do kódu zanést třeba i nějaké další chyby. [13]

Výhodnější možností je proto generování dat pro profilovací analýzu automatizovaným způsobem buďto samotným překladačem nebo specializovaným programem. Tyto nástroje jsou založeny na podobném principu, jak jsem popisoval. Rozdíl je ale v tom, že nyní zůstává zdrojový kód nedotčený, všechny logovací výpisy se do programu přidávají automaticky při kompilaci. [13]

V projektu se budu zabývat programováním v jazyce C a budu používat překladač GCC. Pomocí tohoto překladače můžeme snadno získávat profilovací data tak, že program přeložíme s přepínačem -p, resp. -pg. Překladač pak automaticky do výsledné aplikace doplní logovací kód, který po spuštění programu vygeneruje tzv. graf volání (call graph) a bude uložen do souboru gmon.out. Pomocí programu prof, resp. gprof si poté můžeme zobrazit výsledky analýzy v čitelné formě. [13]

5.1.1 Kompilace programu pro profilování

Prvním krokem při profilovací analýze je zkompilovat program s povolením pro profilování. Pro povolení profilování použijeme přepínač –pg, když spouštíme kompilaci. Také při linkování je zapotřebí povolit profilování, takže při spuštění linkeru uvedeme také přepínač –pg. Úplně stejný přepínač použijeme při současné kompilaci i linkování. Zde je příklad:

gcc -g -c prog.c utils.c -pg

gcc -o prog prog.o utils.o -pg

Druhou možností je použít jeden příkaz pro současnou kompilaci i linkování s profilováním.

gcc -o prog prog.c utils.c -g -pg

Pokud zkompilujeme pouze nějaké moduly pomocí přepínače –pg, můžeme i tak profilovat program, ale nebudeme mít úplné informace o těch modulech, které byly kompilovány bez povolení profilování. Jedinou informací, kterou o těchto funkcích dostaneme, je celková doba strávená při jejich provádění. Nedostaneme žádné informace o tom, kolikrát byly zavolány, nebo odkud byly volány. [14]

Pokud bychom chtěli provádět profilování line-by-line, musíme specifikovat přepínač –g, který kompilátoru říká, aby do výsledného programu vkládal informace pro debugger. Jsou to symboly v programu, které mapují adresu v programu na řádky ve zdrojovém kódu. [14]

K přepínačům –pg a –g můžeme použít ještě další přepínač –a. Pomocí toho instruujeme program tak, aby prováděl základní počítání bloků. Když takto přeložený program spustíme, bude se do logovacího souboru zaznamenávat, kolikrát byl proveden skok v každém bloku IF, kolik bylo provedeno iterací v každém bloku DO, FOR, WHILE, atd. To dále umožní programu gprof vytvořit komentář ke každému řádku kódu, který nám řekne, kolikrát byl jednotlivý řádek proveden. Takto je možné nalézt ty cykly, které se provádějí nejčastěji, a které by bylo vhodné optimalizovat. [14]

5.1.2 Spuštění profilovaného programu

Pokud máme náš program zkompilovaný pro profilování, je potřeba ho už jen spustit, aby se vygenerovaly informace, které bude potřebovat program GPROF, jednoduše spustíme program s parametry jako obvykle. Program poběží standardně, jako kdyby ani nebyl profilován, bude mít stejný výstup jak je očekávané. Rozdíl bude především v rychlosti běhu programu, program poběží výrazně pomaleji než normálně, protože se musí počítat časy strávené v jednotlivých částech programu, proběhne sběr důležitých dat a zápis do logovacího souboru. [14]

Jak už bylo řečeno, způsob, jak bude program spuštěn, tedy jeho parametry, může mít veliký vliv na vygenerované profilovací informace. Profilovací data budou popisovat běh programu pro specifický vstup programu. Na každá vstupní data může náš program reagovat trochu jinak, spouštět jiné funkce, pracovat rychleji či pomaleji. Měli bychom proto pro profilování zvolit takové parametry programu, které budou co nejvíce objektivní a budou jakýmsi reprezentativním vzorkem pro běžné spouštění programu. Pokud bychom například spustili program pouze pro vypsání jeho nápovědy, v profilovacích datech by byla pouze informace o nějaké funkci HELP() a o ukončovacích funkcích. Z těchto dat bychom nezískali nic užitečného. [14]

Program bude po spuštění zapisovat do souboru jménem gmon.out. Musíme si dát pozor, pokud již takový soubor existuje, protože jeho obsah bude přepsán novými daty. Bohužel nemáme možnost specifikovat jiné jméno cílového souboru, můžeme ale tento soubor přejmenovat, aby nedošlo při dalším spuštění programu k jeho přepsání. Aby byl soubor gmon.out úspěšně zapsán na disk, musí program skončit standardně, návratem z hlavní funkce main() nebo zavoláním funkce exit(). Zavoláním nízko-úrovňové funkce _exit() nedojde ke standardnímu ukončení programu. [14]

5.1.3 GNU GPROF

Nyní, když máme profilovací data uložena v souboru gmon.out, můžeme nyní použít program prof., který nám interpretuje tato data a zobrazí je v čitelné formě na standardní výstup. Typicky si necháme výstup přesměrovat do nějakého souboru pomocí “>“ a ten potom budeme procházet a analyzovat.
gprof se spouští tímto způsobem:

gprof options [executable-file [profile-data-files...]] [> outfile]

Hranaté závorky reprezentují volitelné parametry. Všimněme si dále, že gprof bere jako parametr také náš spustitelný program. Pokud nebude zadáno jméno spustitelného souboru, potom bude použit standardně program a.out, pokud nebudou specifikována jména profilovacích souborů, bude použito standardní jméno profilovacího souboru gmon.out. Pokud by některý ze souborů nebyl v požadovaném formátu nebo pokud by profilovací data nebyla určena pro daný spustitelný soubor, pak se zobrazí chybová zpráva. [14]

Je možné zadat více souborů s profilovacími daty, potom statistiky ze všech souborů budou brány dohromady. Jedna ze zobrazených tabulek může vypadat takto:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  0.00      0.00     0.00      632     0.00     0.00  function1
  0.00      0.00     0.00      341     0.00     0.00  function2
  0.00      0.00     0.00      281     0.00     0.00  isNULL
  0.00      0.00     0.00        1     0.00     0.00  readParam
```

Význam jednotlivých sloupců je následující.

% time

Čas strávený v této funkci vyjádřený procentuálně k celkovému času programu

cumulative seconds

Čas strávený v této funkci vyjádřený v sekundách včetně součtů časů, které program strávil ve funkcích volaných z této funkce.

self seconds

Čas strávený výhradně v této funkci vyjádřený v sekundách. Do tohoto času se nezapočítávají časy funkcí, které jsou z této funkce volány.

calls

Součet všech volání této funkce.

self ms/call

Průměrný počet milisekund strávených při vykonávání kódu této funkce při jednom zavolání.

total ms/call

Průměrný počet milisekund strávených v této funkci při jednom zavolání. Údaj zahrnuje i čas strávený ve funkcích volaných z této funkce.

6 Knihovna pro práci s KA

Nyní se zaměříme na samotné algoritmy determinizace a minimalizace konečných automatů a jejich implementaci do knihovny. Knihovna bude implementována v jazyce C. Pro oba uvedené algoritmy budou navrženy možné způsoby řešení. Nejprve bude daný algoritmus rozebrán v teoretické úrovni. Budou navrženy možnosti jeho řešení na teoretické úrovni a poté bude následovat praktické řešení na dnešních výpočetních strojích. Až bude algoritmus napsán v jeho sekvenční verzi, bude muset být analyzován a upraven tak, aby bylo možno ho co nejefektivněji převést do paralelní verze. Bude zkoumáno zrychlení oproti sekvenční verzi a účinnost paralelního algoritmu.

Při řešení paralelních částí jednotlivých funkcí bude použita technologie SSE2, která nám umožňuje paralelizovat program na úrovni instrukcí. Tuto technologii podporují procesory řady Pentium, takže se v projektu omezím pouze na tyto stroje. Abychom mohli psát optimální programy, musíme si vybrat hardware, pro který budeme daný program vyvíjet. Výhodou nám bude rychlejší běh a rychlejší zpracování programu, nevýhoda samozřejmě je, že daný program nebude na jiné platformě správně fungovat nebo poběží pomaleji. Procesory AMD nám sice nabízejí podobnou technologii, která se jmenuje 3DNow, ale tu v tomto projektu nebudu používat, protože nepracuji na počítačích s procesory AMD. Projekt by se pochopitelně dal ještě rozšířit o podporu 3DNow a nahradit tak SSE, kdyby chtěl někdo používat procesory AMD místo Pentium.

SSE nám umožní vykonávat instrukce SIMD (Single Instruction – Multiple Data), kdy my zadáme jednu instrukci, která se provede nad několika daty současně v jednom kroku. Takto můžeme provést porovnání dvou velkých vektorů nebo jejich vzájemné sečtení, odečtení, bitové operace s nimi a další. O SSE a jeho efektivitě či vhodnosti ho používat bude ještě jedna kapitola. Tam se dozvíme, pro které případy bude vhodné používat SSE instrukce, a kde je zase naopak výhodnější použít optimalizované knihovní funkce jazyka C.

Z důvodu efektivnosti je nutné dobře navrhnout strukturu a způsob uložení automatů v paměti počítače. A ne jen samotných automatů. Všechny pomocné struktury budou muset být navrženy tak, aby byl přístup k nim co nejrychlejší. Takže všechny hodnoty, které se budou číst sekvenčně by měly být v paměti uloženy vždy těsně za sebou, to nejen kvůli SSE, ale i pro rychlejší zpracování načítaných hodnot v procesoru atd. Víme, že sekvenční přístup do paměti je mnohem rychlejší než náhodný přístup, protože při sekvenčním čtení mohou být data z paměti načtena dopředu jako celek.

Nakonec budou následovat testy, které nám řeknou, zda nám naše řešení přináší zrychlení, a za jakých podmínek. Provedeme porovnání všech algoritmů, sekvenčních i paralelních verzí. Rozebereme také paměťové náročnosti všech algoritmů a určíme, který z algoritmu by bylo nejlepší kdy použít v závislosti na použitém hardwaru.

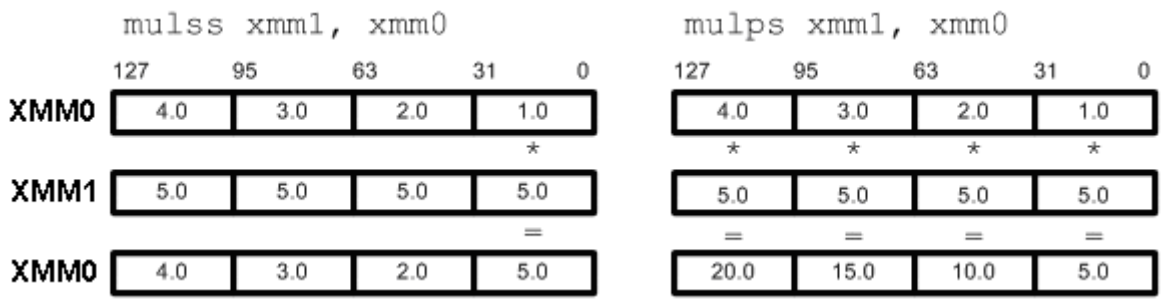
6.1 SSE (Streaming SIMD Extentions)

SIMD (Single Instruction, Multiple Data) je technologie v procesoru Intel, která dovoluje jednomu procesu běžícímu na jednom procesoru vykonávat paralelně jednu instrukci na množině dat (4 float operandy). SSE byla především navržena pro práci v multimediálních aplikacích, kde se zpracovávají vektory čísel, a nad celými vektory má být provedena jedna operace (např. sečtení dvou vektorů, nebo bitový součin, nebo i složitější výpočty). Použití SSE může mít v ideálním případě za následek 4-násobné zrychlení celého výpočtu. Ovšem ne vždy a všude je vhodné SSE použít. Ukážeme si nejdříve, co nám vlastně dovolí SSE, jak se používá, a poté bude následovat ukázka použití. Porovnáme výsledky funkcí psaných v SSE a funkcí počítající postupně v cyklu for, dále porovnáme rychlost výpočtu oproti standardním knihovním funkcím v programovacím jazyce C.
SSE nám definuje 8 nových 128-bitových registrů (xmm0 - xmm7). Tyto registry jsou použity pouze pro SIMD výpočty. Jelikož je jejich velikost rovna 128 bitů, můžeme v nich ukládat až 4 32-bitové hodnoty v plovoucí řádové čárce (4 floaty).

Obrázek 12 - XMM register

6.1.1 Skalární a vektorové instrukce

V SSE jsou definovány dvě skupiny operací; skalární a vektorové. Skalární instrukce pracují pouze s jedním (nejméně významným) 32-bitovým datovým segmentem (data0). Vektorové instrukce počítají se všemi 4 segmenty paralelně v jedné instrukci. SSE instrukce poznáme podle přípony. Přípona -ss se používá u skalárních operací (Single Scalar) a -ps používáme pro vektorové instrukce (Parallel Scalar). Viz. Obr. 13. Výsledek operace se uloží do cílového registru xmm0. Zatímco při paralelní instrukci je přepsán celý registr, tak u skalární instrukce se přepíše pouze poslední segment, ostatní 3 segmenty zůstávají nezměněny.



Obrázek 13 - Skalární a vektorová instrukce

6.1.2 Načítání dat

První věc, kterou potřebujeme vědět, je způsob, jakým se načítají data z paměti do xmm registrů, a dále jak dostat výsledky zpět do naší aplikace poté, co nad nimi provedeme SIMD operace. Instrukce, které nám načítají data z a do paměti můžou být opět buď skalární nebo vektorové:

MOVSS: kopíruje pouze jeden 32-bitový float register

MOVLPS: kopíruje dva 32-bitové float registry (do dvou nejméně významných segmentů)

MOVHPS: kopíruje dva 32-bitové float registry (do dvou nejvíce významných segmentů)

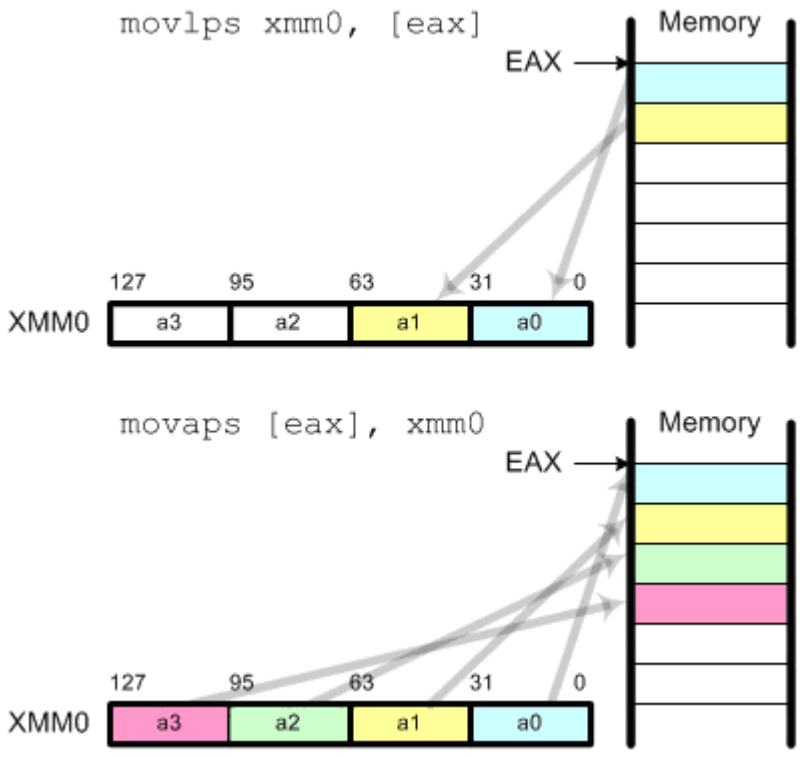
MOVAPS: kopíruje 4 zarovnané float data do celého xmm registru (rychle)

MOVUPS: kopíruje 4 nezarovnané float data do celého xmm registru (pomalu)

MOVHLPS: kopíruje dva nejvíce významné segmenty do nejméně významných

MOVLHPS: kopíruje dva nejméně významné segmenty do nejvíce významných

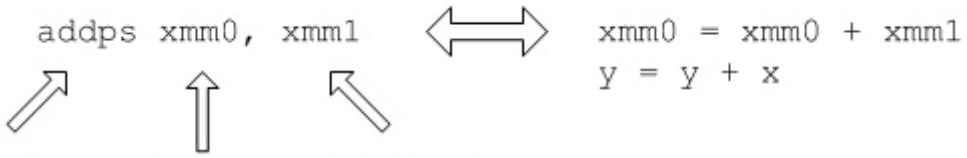
Nejdůležitějšími a nejpoužívanějšími instrukcemi jsou MOVAPS a MOVUPS. Důležitým rozdílem mezi těmito dvěma instrukce mi je ten, že MOVAPS požaduje, aby adresa operandu v paměti byla zarovnaná na 16 bytů! Jinak tato instrukce nepracuje správně a způsobuje pád aplikace. Mně také trvalo nějakou dobu, když jsem takový program ladil, než jsem odhalil příčinu problému. Pokud nemáme data v paměti zarovnána na 16 bytů, můžeme použít instrukci MOVUPS, která je však 4-krát pomalejší než MOVAPS, protože načítá z paměti všechny 4 hodnoty postupně. Proto bych doporučil používat zarovnanou paměť, aby i načítání dat z paměti do XMM registru bylo urychleno. Operandy u instrukcí MOVHLPS a MOVLHPS musí být XMM registry.



Obrázek 14 - Načítání dat do XMM registrů

6.1.3 Aritmetické instrukce

Všechny aritmetické operace pracují se dvěma operandy (s registry nebo přímo s pamětí). Po provedení operace je výsledek vrácen do prvního registru. Zdrojový operand může být XMM registr nebo paměť, avšak cílový operand může být pouze XMM registr.



Obrázek 15 - Příklad SSE instrukce

Typ instrukce	Skalární instrukce	Vektorová instrukce
$y = y + x$	ADDSS	ADDPS
$y = y - x$	SUBSS	SUBPS
$y = y \times x$	MULSS	MULPS
$y = y \div x$	DIVSS	DIVPS
$y = \frac{1}{x}$	RCPSS	RCPPS
$y = \sqrt{x}$	SQRTSS	SQRTPS
$y = \frac{1}{\sqrt{x}}$	RSQRTSS	RSQRTPS
$y = \max(y, x)$	MAXSS	MAXPS
$y = \min(y, x)$	MINSS	MINPS

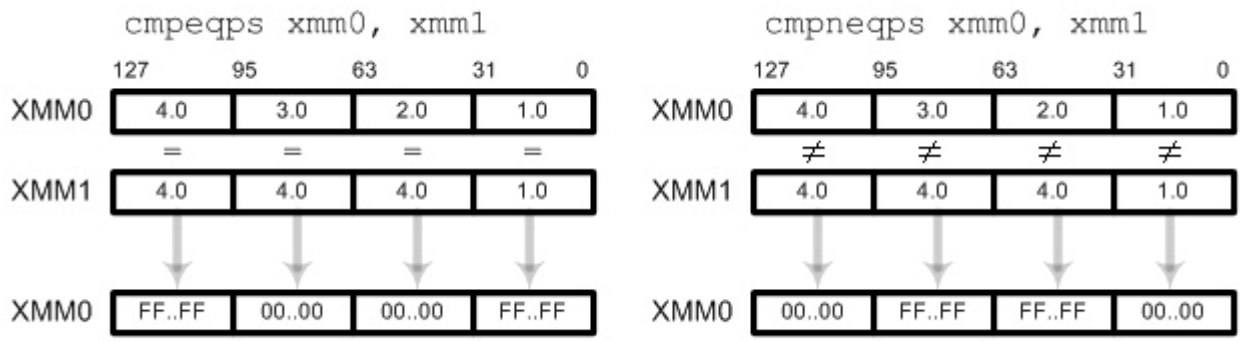
Tabulka 6 - Aritmetické instrukce

6.1.4 Instrukce pro porovnávání

Porovnávací instrukce porovnávají 2 operandy a poté nastavují segment cílového operandu do samých jedniček (pokud porovnání vrátí true) nebo do samých nul (pokud porovnání vrátí false). Zdrojový operand může být XMM registr nebo paměť, avšak cílový operand může být pouze XMM registr. Tyto instrukce mají bohužel jednu nevýhodu. Po provedení porovnání není možné hned provádět podmíněné skoky, jak jsme zvyklí z assembleru. Musíme si nejdříve výsledky porovnání uložit do nějaké pomocné proměnné a tuto proměnnou znovu otestovat standardními porovnávacími instrukcemi. Teprve poté můžeme provádět podmíněné skoky.

Typ instrukce	Skalární instrukce	Vektorová instrukce
x = y, x ≠ y	CMPEQSS, CMPNEQSS	CMPEQPS, CMPNEQPS
x < y, x ≮ y	CMPLTSS, CMPNLTSS	CMPLTPS, CMPNLT
x ≤ y, x ≰ y	CMPLESS, CMPNLESS	CMPLEPS, CMPNLEPS

Tabulka 7 - Instrukce pro porovnávání



Obrázek 16 - Instrukce pro porovnávání

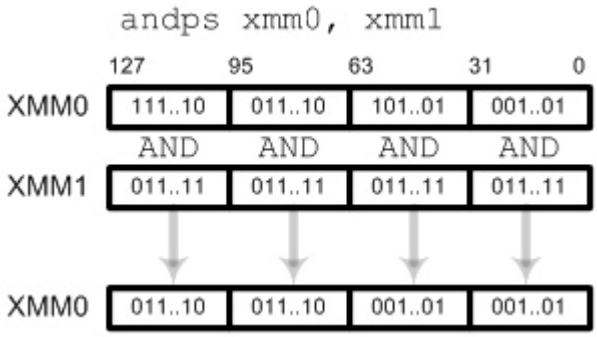
6.1.5 Bitové logické instrukce

Logické instrukce provádí logické operace nad vektorovými float čísly. Jejich typické použití je pro negaci čísla nebo spočítání absolutní hodnoty. Avšak dají se využít, pokud pomocí bitového pole reprezentujeme množinu. Potom můžeme tyto množinové operace provádět ještě 4-krát rychleji.

Operace	Instrukce
AND	ANDPS
OR	ORPS
XOR	XORPS
AND NOT	ANDNPS

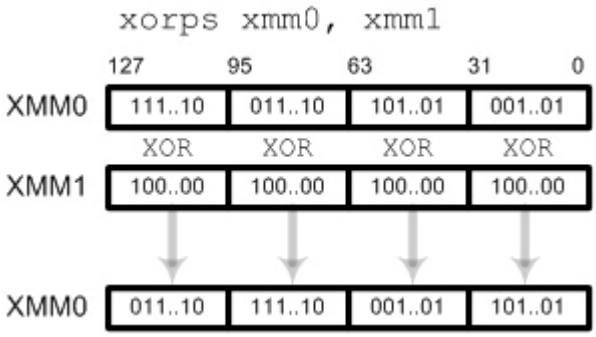
Tabulka 8 - Bitové logické instrukce

Pro výpočet absolutní hodnoty float čísla můžeme použít masku 0x7FFFFFFF a s touto maskou provést logický součin hodnot. Jelikož float čísla mají na prvním bitu znaménko, takto se každé znaménko vynuluje a dostaneme absolutní hodnotu 4 hodnot současně.



Obrázek 17 - Výpočet absolutní hodnoty čtyř floatů

Obdobným způsobem se dá dělat také negace čísel, kdy pomocí masky 0x80000000 a operace XOR změní první bit čísla a tím se změní znaménko.



Obrázek 18 - Negace čtyř float hodnot

6.1.6 Práce s pamětí CACHE

Pomocí SSE můžeme za běhu programu nahrát do paměti cache data, která budeme v budoucnu potřebovat. Nahrávání dat do cache je transparentní a probíhá současně s prováděním dalších instrukcí. Takto můžeme mít připravena data v cache ještě před tím, než je budeme skutečně potřebovat. Takto se minimalizuje doba potřebná pro načítání dat z paměti. Instrukce, které se pro to využívají jsou PREFETCHNTA, PREFETCH0 nebo PREFETCH1:

PREFETCH0: přesunout data z paměti do L1 a L2 cache.

PREFETCH1: přesunout data z paměti pouze do L2 cache.

PREFETCHNTA: přesunout netemporální data z paměti přímo do L1 cache (tímto se obejde L2). Tato data v paměti však musí být opět zarovnána na 16 bytů jako u instrukce MOVAPS.

U procesorů AMD AthlonXP, Intel Pentium4 nebo vyšších jsou data do paměti cache předpřipravována automaticky, proto není nutné volat tyto instrukce ručně v kódu.

Instrukce pro ukládání nebo načítání netemporálních dat načítají údaje přímo do paměti bez aktualizace cache. To minimalizuje znečištění mezipaměti a zvyšuje propustnost sběrnice mezi cache a XMM registry. Netemporální znamená, že data jsou používána nepravidelně v dlouhých intervalech (vztaženo jednou a ne opakovaně v nejbližší budoucnosti), například vertex data ve 3D grafice jsou vytvořeny znovu pro každý snímek.

MOVNTPS: přesunout 4 netemporální floating-point prvky z XMM registru do paměti přímo a obejít cache. Adresa paměti musí být zarovnána na 16 bytové hranici.

MOVNTQ: přesunout netemporální čtyř-slovo (64 bitů) z XMM registru do paměti a obejít cache.

```
MOVNTPS [edi], xmm0
MOVNTQ [edi], xmm0
```

6.2 Funkce pro práci s SSE

V našich programech můžeme používat assembler pro vkládání různých optimalizovaných částí kódu. Jazyk C nám však nabízí řadu SSE funkcí, které tuto činnost provedou za nás, které nám dovolí psát SSE instrukce jako standardní funkce jazyka C. Tyto funkce jsou poté převedeny na assemblerovské instrukce, které známe z minulé podkapitoly. Při používání SSE jsem navíc zjistil, že instrukce, které jsem napsal přímo v assembleru už se dále neoptimalizují a zůstanou v programu tak, jak je tam napíšeme.

V jazyce C máme programovací rozhraní pro použití SSE instrukcí. SIMD vektorový registr XMM je v tomto rozhraní popsán pomocí speciálního 128 bitového datového typu. Dále jsou definovány funkce, které se používají pro SIMD operace nad těmito proměnnými. Intel definuje tři základní datové typy pro práci s SSE v C. Jsou to:

__m128i – pro práci s vektorem celých čísel integer.

__m128 – pro práci se čtyřmi float čísly.

__m128d – pro práci se dvěma 64 bitovými double hodnotami.

Tyto datové typy jsou přenositelné jak na překladače GNU C, tak na Intel C překladače i na další překladače podporující operační systémy Windows.

Intel dále definuje množinu funkcí pro programování SSE v C. Tyto funkce poznáme podle prefixu **_mm_**, dále je napsána operace, kterou provádíme (např. **and**) a jako příponu má funkce poznamenáno, s jakým datovým typem se vlastně pracuje (např. **si128**). Celá funkce potom může vypadat **_mm_and_si128**(xmm0, xmm1).

Ve svém projektu budu pracovat pouze s celočíselnými hodnotami, takže budu používat pouze funkce mající jednu z následujících přípon:

-epi# XMM (128-bitů) vektor obsahující #-bitové znaménkové celé číslo

-epu# XMM (128-bitů) vektor obsahující #-bitové bezznaménkové celé číslo

-si128 XMM (128-bitů) celý vektor jako jedna hodnota

Pro použití těchto funkcí bude potřeba do programu vložit hlavičkové soubory podle požadované verze SSE. Já budu používat SSE2, takže využiji hlavičkový soubor <emmintrin.h>.

Při kompilaci je nutné zadat přepínač **–sse2**, jinak skončí překlad s chybou.

Kompletní množina všech dostupných operací je popsána detailně v manuálu od firmy Intel: Intel Architecture Software Developer's Manual Volume 2.

6.3 Hledání možného zrychlení

Když už nyní víme, jak pracuje SSE, jaké používá instrukce a jak se s nimi dá pracovat v jazyce C, můžeme vyzkoušet a porovnat rychlost běhu programu při použití SSE a při použití některých vestavěných knihovních funkcí jazyka C. V rámci zrychlování paralelních algoritmů jsem zkoumal různé varianty výpočtu. Uvidíme tak, jaké instrukce a v jakém případě bude možno využít i v mém projektu. Provedu tři testy. Při prvním testu budu provádět porovnávací operaci nad dvěma vektory. Možnosti, jakými to budu provádět budou standardní knihovní funkce memcmp(), SSE a cyklus for. Jako druhý provedu test bitových operací nad dvěma vektory. Jako poslední se pokusím v SSE napsat kopírovací funkci a porovnám ji se standardní funkcí memcpy() a opět s iteračním cyklem for.

Pro tyto testy jsem vytvořil zdrojový kód uložený v souboru SSE.c. V tomto souboru jsou napsány funkce pro porovnávání, kopírování dvou vektorů a logický součin dvou vektorů (využívající SSE nebo cyklus for). Dále se zde nachází hlavní funkce main(), která provede testy, a na standardní výstup bude vypisovat doby trvání jednotlivých funkcí. Pro získání doby trvání používám funkci gettimeofday(), takže doba trvání jednotlivých výpočtů bude spočtena s přesností na mikrosekundy.

6.3.1 Funkce pro kopírování dvou vektorů

V souboru SSE.c jsem napsal 3 funkce, které kopírují vektor celých neznaménkových čísel do druhého. První funkce je memcpy_for, která kopíruje pole standardně v cyklu for:

```
inline void memcpy_for (void* Dst, void* Src, size_t Size) {
   for ( unsigned int i = 0; i < Size/4; ++i )
      *((unsigned long int*)(Dst)+i) = *((unsigned long int*)(Src)+i);
}
```

Algoritmus memcpy_for

Druhá funkce memcpy_sse2 používá už SSE instrukce a kopíruje také v cyklu for vždy 4 hodnoty naráz. K tomuto se využívají instrukce **_mm_load_si128**, která načte z paměti 128 bitů do xmm registru a potom **_mm_store_si128**, která zapíše data zpět z registru do paměti. Pokud není počet bytů násobkem 16, tak je zbytek bytů zkopírován také cyklem for. Tato funkce by se dala použít pro kopírování libovolného typu hodnot, protože se vždy zkopírují všechny byty, i když nejsou násobkem 16.

```
inline void memcpy_sse2 (void* Dst, void* Src, size_t Size) {
   size_t size_sse2 = Size/16*16;
   for ( unsigned int i = 0; i < size_sse2; i += 16 ) {
      __m128i xmm0 = _mm_load_si128((__m128i*)((char*)(Src)+i));
      _mm_store_si128((__m128i*)((char*)(Dst)+i), xmm0);
   }
   for ( unsigned int i = size_sse2; i < Size; ++i )
      *((char*)(Dst)+i) = *((char*)(Src)+i);
}
```

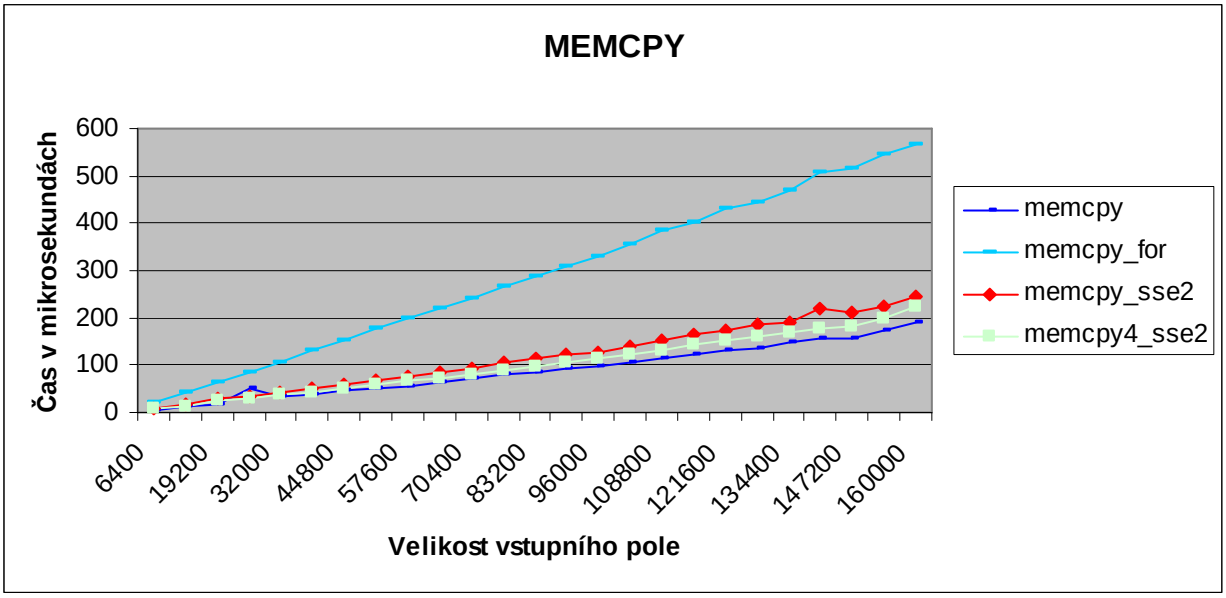
Algoritmus memcpy_sse2

Druhou funkci jsem navíc optimalizoval, aby běžela rychleji. Je známo, že na začátku cyklu for (před každým provedením celého bloku cyklu) je testována podmínka ukončení cyklu. Zde se testuje iterační proměnná **i** s hodnotou velikosti vstupu **size_sse2**. Pokud je tělo cyklu krátké, může toto testování trvat relativně dlouho vzhledem k celému výpočtu. Zde v těle cyklu provádíme pouze jednoduché a rychlé přesuny dat do paměti, žádné složité výpočty, tudíž testování ukončení cyklu má veliký podíl na celé době trvání algoritmu. Vytvořil jsem proto další funkci **memcpy4_sse2**, která v těle cyklu provádí hned 4 přesuny místo jednoho. Takto jsem minimalizoval počet testování konce cyklu pouze na čtvrtinu. Tím bych měl dosáhnout optimálního zrychlení cyklu for.

```
inline void memcpy4_sse2 (void* Dst, void* Src, size_t Size) {
   size_t size_sse2 = Size/64*64;
   for ( unsigned int i = 0; i < size_sse2; i += 64 ) {
      __m128i xmm0 = _mm_load_si128((__m128i*)((char*)(Src)+i));
      __m128i xmm1 = _mm_load_si128((__m128i*)((char*)(Src)+i+16));
      __m128i xmm2 = _mm_load_si128((__m128i*)((char*)(Src)+i+32));
      __m128i xmm3 = _mm_load_si128((__m128i*)((char*)(Src)+i+48));
      _mm_store_si128((__m128i*)((char*)(Dst)+i), xmm0);
      _mm_store_si128((__m128i*)((char*)(Dst)+i+16), xmm1);
      _mm_store_si128((__m128i*)((char*)(Dst)+i+32), xmm2);
      _mm_store_si128((__m128i*)((char*)(Dst)+i+48), xmm3);
   }
   for ( unsigned int i = size_sse2; i < Size; ++i )
      *((char*)(Dst)+i) = *((char*)(Src)+i);
}
```

Algoritmus memcpy4_sse2

Test rychlosti proběhl na jednoprocesorovém počítači Intel Centrino 1,73GHz s operačním systémem CentOS 5. Pro přesnější výsledky jsem měřil čas v cyklu několikrát a poté jsem spočítal průměrnou hodnotu. Pokud bych nechal každou funkci spustit pouze jednou, tak by výsledky nemusely odpovídat přesně skutečnosti. To je vlivem přepínání kontextu, kdy může být náš program pozastaven, a místo něj může běžet úplně jiný program. Potom by se do výsledného času negativně promítnuly všechny časy všech běžících procesů. Z následujícího grafu můžeme přečíst průměrnou dobu trvání výpočtu v závislosti na velikosti vstupního pole.



Nejdelší dobu pochopitelně trvala funkce memcpy_for, protože prováděla kopírování po jednotlivých hodnotách. Mnohem lépe na tom byly funkce používající SSE, které přinesly asi dvojnásobné zrychlení. Nejrychlejší však byla optimalizovaná standardní funkce memcpy.

6.3.2 Funkce pro logický součin dvou vektorů

V souboru SSE.c jsem dále napsal 3 funkce, které provádí logický součin hodnot jednoho vektoru s hodnotami ve druhém vektoru. První funkcí je and_for, která provádí výpočet logického součinu (bez pomoci SSE) standardně v cyklu for:

```
inline void and_for (void* Dst, void* Src1, void* Src2, size_t Size) {
    for ( unsigned int i = 0; i < Size/4; ++i )
        *((unsigned long int*)(Dst)+i) = (*((unsigned long int*)(Src1)+i)) &
                                         (*((unsigned long int*)(Src2)+i));
}
```

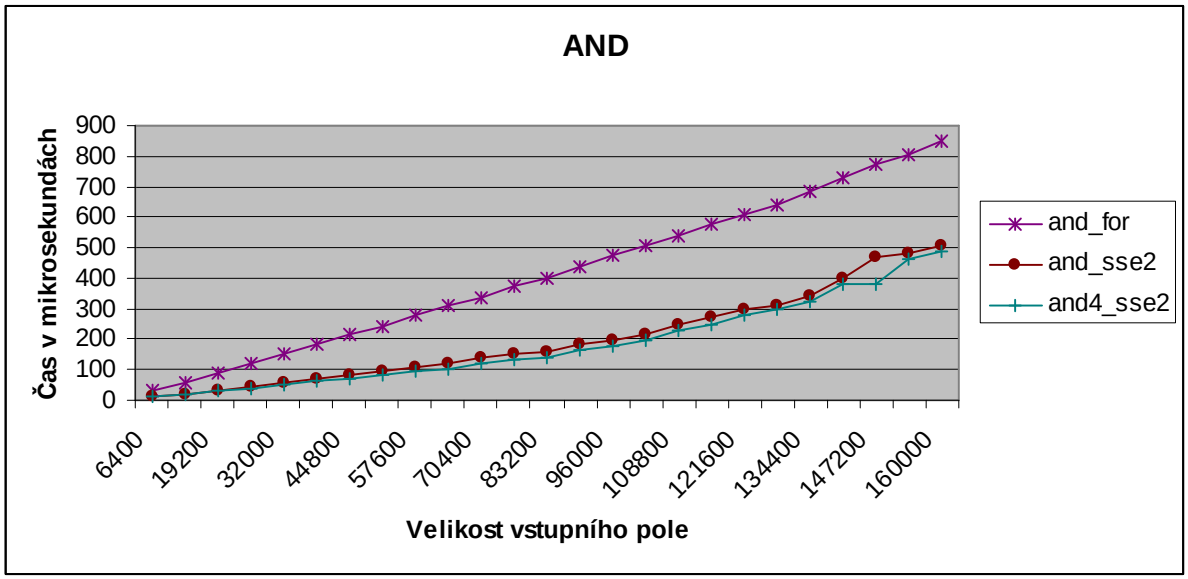
Algoritmus anf_for

Obdobně, jako tomu bylo v předešlé podkapitole, i pro funkci logického součinu jsem napsal dvě verze funkce využívající SSE. První verze **and_sse2** násobí v těle cyklu pouze jednu 128 bitovou hodnotu s druhou 128 bitovou hodnotou. Provádí se zde tedy načtení dvou dat do xmm registrů funkcí **_mm_load_si128**, následuje SSE instrukce pro bitový součin **_mm_and_si128** a nakonec je výsledek operace uložen zpět do paměti instrukcí **_mm_store_si128**. Vše je opět v jednom cyklu for:

```
for ( unsigned int i = 0; i < size_sse2; i += 16 ) {
    __m128i xmm0 = _mm_load_si128((__m128i*)((char*)(Src1)+i));
    __m128i xmm1 = _mm_load_si128((__m128i*)((char*)(Src2)+i));
    xmm0 = _mm_and_si128(xmm0, xmm1);
    _mm_store_si128((__m128i*)((char*)(Dst)+i), xmm0);
}
```

Hlavní část algoritmu and_sse2

Optimalizovaná verze tohoto algoritmu je ve funkci **and4_sse2**. Opět jsem se snažil minimalizovat počet testování konce cyklu. Zmenšil jsem počet testování a následných podmíněných skoků na čtvrtinu tím, že jsem využil všech 8 dostupných registrů xmm. Nová funkce vznikne z funkce předchozí, ta se v těle cyklu rozšíří o několik dalších instrukcí. Nejprve se čtyřikrát použije instrukce pro načtení dat z prvního vektoru, potom se načtou data z druhého vektoru. Když máme data připravena ve všech osmi registrech, provede se nad všemi dvojicemi patřičná operace logického součinu. Nakonec se výsledky zapíší zpět do paměti. Je zřejmé, že tělo tohoto cyklu bude výrazně delší než testování konce, tudíž bude režie tohoto cyklu mnohem menší. Uvidíme dále.



Test rychlosti proběhl na stejném jednoprocesorovém počítači Intel Centrino 1,73GHz s operačním systémem CentOS 5. Pro dosažení přesnějších výsledků jsem měřil čas v cyklu několikrát a poté jsem spočítal průměrnou hodnotu. Z předchozího grafu můžeme přečíst průměrnou dobu trvání výpočtu v závislosti na velikosti vstupního pole. Jako v předchozím případě, ani teď nedosahujeme použitím SSE čtyřnásobného zrychlení, ale pouze dvojnásobného. Ale i přes to je to zrychlení patrné. Použití SSE by bylo v tomto případě tedy jistě vhodné, a to především proto, že neexistuje žádná standardní funkce jazyka C, která by nám počítala bitový součin dvou vektorů. Právě z tohoto důvodu byly instrukce SSE do procesoru přidány. Právě pro podobné výpočty prováděné nad vektorem čísel byla technologie SSE navržena. Nejedná se pochopitelně pouze o operaci bitového součinu ale obecně o jakýkoliv výpočet nad velkým vektorem čísel. V minulém případě, když jsme použili SSE pouze k načtení dat, nevyužili jsme plně celý potenciál této technologie. Navíc existovala optimalizovaná standardní funkce jazyka C, která byla rychlejší. Ale v tomto případě se použití SSE vyplatí.

6.3.3 Funkce pro porovnávání

Naposledy budu zkoumat porovnávací instrukce v SSE. Ve zdrojovém souboru SSE.c jsem k tomuto napsal tři porovnávací funkce; první funkce porovnává dva vektory zase pomocí jednoho iterovaného cyklu for, druhá a třetí funkce používají SSE instrukce. Porovnávací funkce testuji proto, že je tu v SSE trochu odlišný způsob, jak provádět podmíněné skoky na základě porovnávání.

První funkci **memcmp_for** už nebudu rozepisovat, podíváme se blíže na další dvě.

Jak tedy provedeme podmíněný skok na základě porovnávacích instrukcí SSE? V kapitole 6.1.4 jsme si ukázali, že výsledek porovnání dvou xmm registrů zapisuje buď samé jedničky nebo samé nuly do cílového registru. Nikam se ale nezapisují žádné příznaky, takže další instrukce procesoru nemůže být podmíněný skok. V SSE se toto řeší pomocí instrukcí **MOVMSKPD**, **MOVMSKPS** nebo **PMOVMSKB**, které kopírují první bit z každého datového **double**, **float** nebo **int** elementu a spojí je dohromady do **dvou**, **čtyř** nebo **šestnácti** bitů do jednoho registru. Tento registr, ve kterém je uloženo bitové pole můžeme potom testovat a provádět pomocí něj podmíněné výrazy. Stačí použít správnou masku a můžeme testovat kterýkoliv datový element. Následující funkce se jmenuje **memcmp_sse2**. Pro porovnání celočíselných hodnot se v SSE použije instrukce _mm_cmpeq_epi8 a její výsledek se poté předá funkci _mm_movemask_epi8. Takto se provede vlastně paralelní porovnání 16 bytových hodnot a dále se výsledky porovnání uloží do registru jako nejméně významných 16 bitů. Maskou 0xFFFF můžeme tento registr otestovat, abychom zjistili, zda porovnání bylo ve všech případech úspěšné. Pokud bylo porovnání neúspěšné, vracím hodnotu 0.

```
for ( unsigned int i = 0; i < size_sse2; i += 16 ) {
    __m128i xmm0 = _mm_load_si128((__m128i*)((char*)(Src)+i));
    __m128i xmm1 = _mm_load_si128((__m128i*)((char*)(Dst)+i));
    xmm0 = _mm_cmpeq_epi8(xmm0, xmm1);
    register int mask = _mm_movemask_epi8(xmm0);
    if ( mask != 0xFFFF )
        return 0;
}
```

Hlavní část algoritmu memcmp_sse2

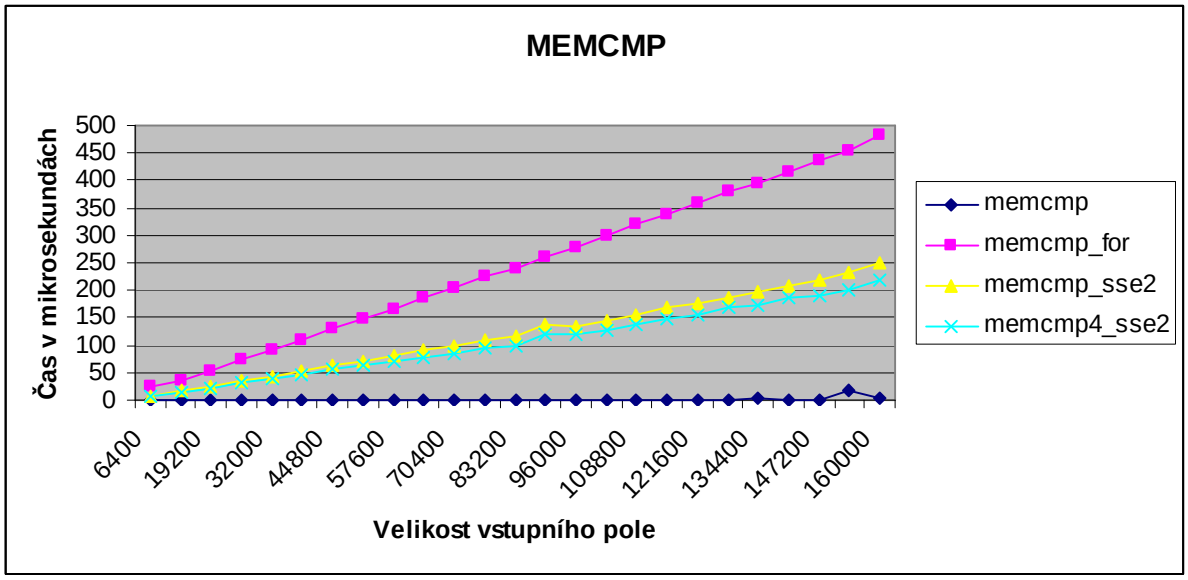
Jak je vidět, instrukce pro provedení podmíněného skoku v SSE skutečně chybí. Použitím dalšího registru, naplněním jeho obsahu maskou a teprve poté další porovnávací instrukcí se velmi zvyšuje režie podmíněných skoků v SSE. Pokud bychom chtěli provádět podmíněný skok na základě pouze jedné proměnné, potom by k tomuto bylo SSE naprosto nevhodné. Tento kód nám může přijít vhod např. zde ve funkci porovnávání dvou velkých vektorů, kdy při první neshodě ukončujeme činnost funkce, a vracíme hodnotu neúspěchu.

Tuto SSE funkci jsem pochopitelně ještě upravil, aby využívala všech 8 registrů v každé iteraci cyklu. Vznikla tak poslední funkce **memcmp4_sse2**, která porovnává v každé iteraci čtyři dvojice hodnot.

```
for ( unsigned int i = 0; i < size_sse2; i += 64 ) {
    __m128i xmm0 = _mm_load_si128((__m128i*)((char*)(Src)+i));
    ... ... ...
    __m128i xmm7 = _mm_load_si128((__m128i*)((char*)(Dst)+i+48));
    xmm0 = _mm_cmpeq_epi8(xmm0, xmm1);
    xmm2 = _mm_cmpeq_epi8(xmm2, xmm3);
    xmm4 = _mm_cmpeq_epi8(xmm4, xmm5);
    xmm6 = _mm_cmpeq_epi8(xmm6, xmm7);
    register int mask1 = _mm_movemask_epi8(xmm0);
    register int mask2 = _mm_movemask_epi8(xmm2);
    register int mask3 = _mm_movemask_epi8(xmm4);
    register int mask4 = _mm_movemask_epi8(xmm6);
    mask1 &= mask2;    mask3 &= mask4;    mask1 &= mask3;
    if ( mask1 != 0xFFFF )
        return 0;
 }
```

Hlavní část algoritmu memcmp4_sse2

Nejprve se čtyřikrát použije instrukce pro načtení dat z prvního vektoru, potom se načtou data z druhého vektoru. Když máme data připravena ve všech osmi registrech, provede se nad všemi dvojicemi patřičná operace porovnání. Zde musíme z každého porovnání přečíst masku a uložit si ji do pomocného registru. Všechny pomocné registry nakonec spojíme do jednoho (pomocí bitového součinu) a tento registr potom můžeme otestovat. Zjistíme, zda všech 16 hodnot úspěšně prošlo testem ekvivalence. V záporném případě vrátíme hodnotu neúspěchu a algoritmus skončí.



Jak jsem předpokládal, nejdelší dobu trvala funkce memcmp_for, protože porovnávala hodnoty po jedné, ale na druhou stranu se zde projevilo zřetězené zpracování procesorem, takže celková doba nebyla 4-krát pomalejší než při použití SSE, nýbrž pouze dvakrát. I funkce memcmp4_sse2 vykazuje nepatrné zrychlení oproti funkci memcmp_sse2, je to tím, že funkce memcmp_sse2 provádí v cyklu pouze jedno porovnání a dále provádí 4-krát více testů na konec cyklu. Nejvíce mě však překvapila standardní funkce memcmp, která se ukázala jako nejrychlejší řešení. Její průběh není konstantní, jak by se možná zdálo z grafu, také má lineární průběh, avšak úhel naklonění této úsečky je velmi malý. Mám dvě vysvětlení tohoto jevu. Buď je tato funkce natolik rychlá a optimalizovaná, nebo je možné, že překladač udělá nějaké optimalizace, které funkci memcmp spustí pouze jednou (přitom ale já ji volám v cyklu několikrát), dále je možné, že funkce memcmp si sama ohlídá, že tato 2 pole už porovnávala, a pouze vrací už předem připravený výsledek. Zkoušel jsem funkci memcmp i na různých počítačích, vždy s podobným výsledkem. Záleží tedy na konkrétním překladači a optimalizátoru, jak bude ve skutečnosti náš program rychlý.

Závěrem těchto tří testů je poznatek, že standardní funkce jazyka C jsou už optimalizované a vyladěné pro nejrychlejší běh našich programů, proto je velmi moudré je používat. Pokud nám některé funkce pro práci s velkými poli chybí, potom můžeme úspěšně použít SSE instrukce namísto standardního cyklu.

6.4 Determinizace NKA

V této kapitole se už budeme věnovat problému determinizace NKA a jeho paralelní implementaci do knihovny. V kapitole 2.7 jsme měli formálně zapsaný algoritmus. Nyní se ho pokusím přepsat do podoby pseudokódu, abychom ho mohli v případě potřeby naimplementovat v jakémkoliv paralelním programovacím jazyce. Následující kód je nejrychlejší implementací, rychleji by už ani determinizace nemohla být provedena.

function ParDetermination(M):

1. Buď $M' = (Q', \Sigma', \delta', q_0', F')$ nový DKA
2. $Q' = 2^Q$ /* $\varnothing$ bude reprezentovat nedefinovaný stav */
3. $\Sigma' = \sum$; $q_0' = \{q_0\}$; $F' = \varnothing$
4. **for each** q **in** Q' **do in parallel**
5. **for each** a **in** Σ **do in parallel**
6. $\delta'(q,a)$ = sjednoceniMnozin(q,a)
7. **if** $q \cap F \neq \varnothing$ **then**
8. $F' = F' \cup \{q\}$
9. **return** M'

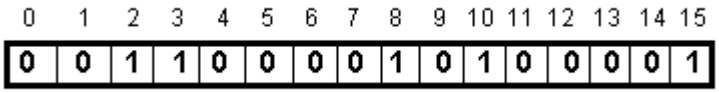
function sjednoceniMnozin(q,a):

1. Buď proměnná sjednoceni nová prázdná množina
2. **for each** qq **in** q **do in parallel**
3. $sjednoceni = sjednoceni \cup \delta(qq,a)$
4. **return** sjednoceni

Nyní si rozebereme časovou složitost a cenu algoritmu ParDetermination. Jediný menší problém by mohla být práce s množinami. Pokud bychom předpokládali, že operace na množinách budou mít konstantní časovou složitost (vhodnou implementací), potom i časová složitost celého algoritmu by byla O(1) ovšem za cenu P = $Q \cdot \sum \cdot 2^Q$ procesorů a navíc za cenu neúnosně velké paměťové náročnosti. Celková cena by byla také C = $Q \cdot \sum \cdot 2^Q$, což je ale optimální řešení, protože doba trvání sekvenčního algoritmu je O($Q \cdot \sum \cdot 2^Q$).

Jaká je však skutečná časová a paměťová náročnost práce s množinami? Podíváme se na možnost, jak se dá reprezentovat množina co nejefektivněji.

Pokud známe předem počet prvků, které může množina obsahovat, můžeme celou množinu reprezentovat pomocí bitového pole o velikosti Q bitů. Potom každý bit v tomto poli znamená, zda se daný prvek reprezentovaný tímto bitem v množině nachází, či ne. Logická 1 může reprezentovat přítomnost prvku v množině, logická 0 nepřítomnost.



Obrázek 19 - Množina reprezentovaná bitovým polem

Na obrázku 19 je názorně vidět, jak se dá reprezentovat množina o max. 16 prvcích. Jedná se o množinu čísel {2,3,8,10,15}. Nemusíme tedy použít seznam, který bychom potom museli dlouze procházet. Operace vložení nebo odebrání prvku z takovéto množiny má konstantní časovou složitost ať je bitové pole jakkoliv dlouhé, protože stačí pouze naadresovat daný bit a logickou operací ho buď vynulovat nebo nastavit na 1. Stejně tak zjištění, zda se nějaký prvek v množině nachází, má konstantní časovou složitost, jelikož nám opět stačí naadresovat daný bit a přečíst si pouze jeho hodnotu. Jak tomu bude ale s operacemi průnik a sjednocení množin? Abychom dokázali provést tyto operace v konstantním čase, musela by v počítači být velikost jednoho slova alespoň tak velká jako je bitové pole, potom bychom dokázali pomocí jedné instrukce provést bitový součin nebo bitový součet dvou bitových polí a tím bychom dosáhli operace sjednocení a průniku v čase O(1). V praxi je ovšem tato podmínka nesplnitelná, protože většinou potřebujeme mít mnohem větší bitové pole než nám dovolí velikost jednoho slova v procesoru. My si tak musíme rozdělit celé bitové pole na části o velikosti jednoho slova a na těchto částech spustit operace separátně. Potom doba výpočtu závisí na velikosti bitového pole lineárně. Časová složitost je potom O(N).

Tímto jsme zjistili, že tedy ani doba determinizace NKA nebude v praxi prováděna s konstantní časovou složitostí. Nejrychleji ji tedy zvládneme v lineárním čase O(Q). I tak bude ale cena algoritmu $C = Q \cdot Q \cdot \sum \cdot 2^Q$ optimální.

Tímto jsme si rozebrali algoritmus determinizace NKA pouze v teoretické rovině. Zatím se počet procesorů nezvyšují takovou rychlostí, abychom jich v praxi mohli využívat P = $Q \cdot \sum \cdot 2^Q$. Proto budeme muset algoritmus upravit tak, aby byl zvládnutelný i na dnešních procesorech. Tímto se už dostáváme k samotnému návrhu a implementaci algoritmu do knihovny. Použijeme upravený algoritmus popsán v kapitole 2.7, demonstrovaný na příkladu. V praxi je tento algoritmus nejčastěji využívaný. Já se v něm navíc pokusím nalézt paralelní části, které by se daly provádět na více procesorech současně a tím nalézt paralelní zrychlení výpočtu.

function SeqDetermination(M):

1. Buď proměnná STACK nový inicializovaný zásobník
2. Buď proměnná LIST nový inicializovaný číslovaný indexovatelný seznam
3. Buď $M' = (Q', \Sigma', \delta', q_0', F')$ nový DKA
4. Q' = {0,1}; $\Sigma' = \Sigma$; $q_0' = 1$; $F' = \emptyset$
5. LIST.ADD($\emptyset$); LIST.ADD({ q_0 })
6. STACK.PUSH(q_0')
7. **while not** STACK.EMPTY **do**
8. q = STACK.POP()
9. **for** a = 0 **to** $|\Sigma|$ - 1 **do**
10. tmp = $\emptyset$
11. ***for each*** *qq* ***in*** LIST[q] **do**
12. $tmp = tmp \cup \delta(qq, a)$
13. x = LIST.Find(tmp)
14. **if not** found **then**
15. x = LIST.ADD(tmp) // vložíme nový stav a získáme jeho číslo
16. STACK.PUSH(x)
17. $Q' = Q' \cup \{x\}$
18. **if** $tmp \cap F \neq \emptyset$ **then**
19. $F' = F' \cup \{q\}$
20. $\delta'(q, a)$ = x
21. **return** M'

Nový determinizovaný automat M' bude obsahovat množinu stavů jako bezznaménková čísla počínaje stavem 0, který reprezentuje sink stav až po stav |Q|-1. Při inicializaci algoritmu (řádky 1-6) vložíme do množiny stavů Q' dva stavy. Prvním je již zmíněný sink stav a druhý stav 1, který bude zároveň startovním stavem. Množinu koncových stavů inicializujeme na prázdnou množinu. Seznam LIST nám bude sloužit jako konverzí tabulka mezi množinou stavů ze vstupního NKA a skutečným číslem stavu v novém automatu. Do seznamu LIST si vložíme oba dva stavy, které již automat má, jedná se na nultém indexu o prázdnou množinu reprezentující sink stav a na 1. indexu je to množina obsahující startovní stav vstupního NKA. Dále si do zásobníku STACK uložíme jako první nový startovní stav a algoritmus je inicializovaný a připravený. Takto bude vypadat inicializace algoritmu vždy. Sekvenční i paralelní verze. Teď se můžeme podívat na tělo cyklu while, které nás bude zajímat ze všeho nejvíce. To je srdce celého algoritmu.

Ze zásobníku se budou jeden po druhém vybírat nově vznikající stavy, abychom mohli pro každý vypočítat přechodové funkce pro všechny symboly z abecedy. Vznikne nám tak úplně definovaný DKA. Nyní přijde první část (řádky 10-12), kdy budeme vytvářet množinu všech stavů, do kterých by automat mohl přejít se symbolem a. Ze seznamu LIST si pomocí indexu q vybereme množinu stavů NKA, která přísluší danému stavu q v DKA. A pro všechny stavy v této množině provedeme sjednocení množin následujících stavů. Výsledek bude uložen do proměnné tmp. Nyní přichází nejsložitější část algoritmu. Potřebujeme zjistit, zda nový stav tmp se již v novém DKA nachází, nebo ne (řádek 13). Pokud tento stav ještě neexistuje, tak ho jednoduše přidáme do automatu, do seznamu a vložíme ho do zásobníku. Poté zaznamenáme ještě přechod v novém automatu. Na konci algoritmu vracíme hotový determinizovaný automat.

Nyní se budeme věnovat implementaci samotných funkcí do knihovny. Knihovna se skládá ze dvou hlavních zdrojových souborů. Jsou to FA.h, kde jsou zapsány hlavičky a deklarace jednotlivých datových typů, struktur a funkcí, a dále soubor FA.c, kde je implementace algoritmů a funkcí. Dále tu máme soubor main.c, to je testovací demo soubor, pomocí kterého budeme provádět testy všech knihovních funkcí.

6.4.1 Datové typy NFA a DFA

V souboru FA.h máme definovány datové typy pro oba typy konečných automatů, pro DKA i NKA. Začneme nejdříve strukturou DFA pro DKA, který je o malinko jednodušší:

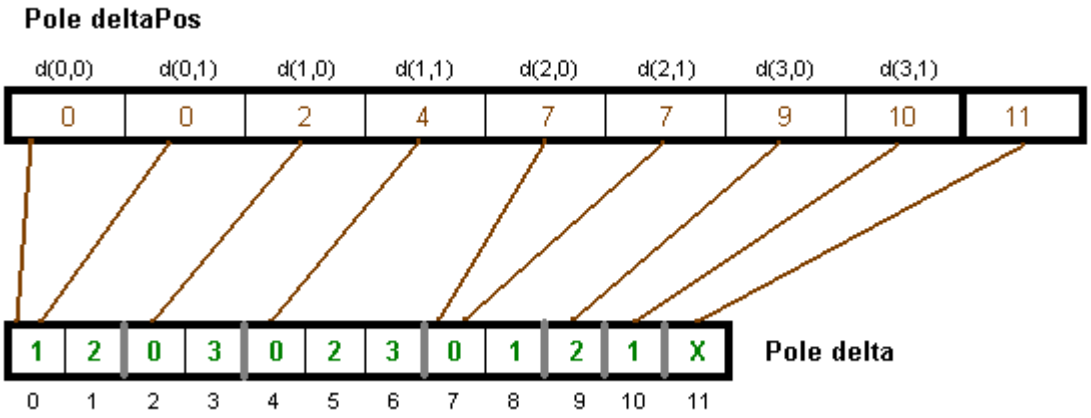
```
typedef struct DFA {
   STATE Q;
   SYMBOL E;
   STATE* delta;
   STATE q0;
   SET F;
} DFA;
```

Definice STATE i SYMBOL nám definují neznaménkové celé číslo. Definice SET definuje ukazatel na bitové pole, které nám bude reprezentovat množinu. Položka Q nám označuje počet stavů; stavy se vždy číslují od 0, proto nemusíme mít množinu čísel. Stejné je to u položky E, která nám označuje počet symbolů abecedy; také jsou číslovány od 0. delta je ukazatel na pole stavů, která nám reprezentuje přechodovou tabulku. Tabulka je paměti uložená jako jednorozměrné pole, takto bude přístup do ní o něco rychlejší, protože nebudeme muset přistupovat k jejím hodnotám přes dva ukazatele. q0 je číslo počátečního stavu a F je množina čísel reprezentovaná bitovým polem.

Dále tu máme datový typ pro uložení NKA, jedná se o strukturu NFA:

```
typedef struct NFA {
   STATE Q;
   SYMBOL E;
   unsigned long int* deltaPos;
   STATE* delta;
   STATE q0;
   SET F;
} NFA;
```

Položky Q a E jsou podobné jako ve struktuře DFA, označují nám počet stavů a počet symbolů a tím nám vlastně říkají, kolik stavů a kolik symbolů je v množině stavů a množině symbolů, každá z nich se počítá od 0. Položky q0 a F jsou také stejné. Drobný rozdíl tu je v reprezentaci přechodových funkcí. Položka delta zde reprezentuje tabulku přechodů. Pole deltaPos je tak velké, aby pokrylo všechny stavy se všemi symboly, a pro každý stav a každý symbol uchovává index do tabulky delta. deltaPos tedy nahrazuje celou přechodovou tabulku v DFA, ale jelikož NFA nemusí přecházet z jednoho stavu a jednoho symbolu pouze do jednoho následujícího stavu, jsou všechny množiny přechodů uloženy v položce delta, a deltaPos vždy ukazuje na začátek konkrétního seznamu stavů. Následující obrázek lépe znázorní použití těchto tabulek.



Obrázek 20 - Reprezentace přechodové tabulky NKA

Všimněme si, že v poli deltaPos nemusíme ukládat dva indexy pro každý stav a symbol. Nemusíme ukládat začátek a konec, stačí nám, když každé pole obsahuje index na začátek svého seznamu hodnot, a tím vlastně ukazuje za konec seznamu předchozího. Dokonce ani nemusíme mít speciální symbol, kterým bychom reprezentovali prázdnou množinu, protože stačí, aby položka další ukazovala na ten samý index jako předchozí. Z toho také plyne, že pole deltaPos bude o jeden prvek větší, než je skutečná velikost tabulky. To nám ovšem nebude vzhledem k velikosti všech tabulek vůbec vadit.

Přechodová tabulka takovéhoto automatu bude vypadat následovně:

	Symbol0	Symbol1
Stav 0	{}	{ 1, 2 }
Stav 1	{ 0, 3 }	{ 0, 2, 3 }
Stav 2	{}	{ 0, 1 }
Stav 3	{ 2 }	{ 1 }

Tabulka 9 - Přechodová tabulka

d(0,0) = {} sice ukazuje na index 0, ale položka následující také ukazuje na stejný index.

d(2,0) = {} z toho samého důvodu.

Pro průchod jednou celou množinou nám stačí jeden jednoduchý cyklus for.

for (i = deltaPos[n]; i < deltaPos[n+1]; ++i)

Pokud dosadíme za deltaPos[0] hodnotu 0 a za deltaPos[1] hodnotu také 0, potom se tento cyklus neprovede ani jednou a tímto provedeme průchod prázdnou množinou.

Jako poslední máme v hlavičkovém souboru ještě deklarace dvou pomocných struktur pro zásobník a pro množinu stavů.

```
typedef struct s_set {
   STATE* States;
   STATE Count;
} t_set;
typedef struct s_stack {
   STATE* States;
   STATE Count;
   STATE Allocated;
} t_stack;
```

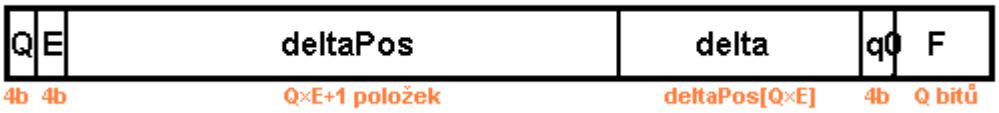
Množina t_set obsahuje pole stavů States, ve kterém budou uloženy všechny stavy jako hodnoty. Kolik těch hodnot dohromady v množině je, to nám říká hodnota Count. Tato struktura byla navržena pro množinu stavů, která se nebude měnit (jednou se naalokuje a vytvoří a už se do této množiny nebudou přidávat ani ubírat žádné prvky), především se využije v seznamu LIST, do kterého se budou pouze zapisovat hodnoty a dále už se nebudou měnit.

Zásobník t_stack obsahuje kromě seznamu všech uložených stavů (stavy se v tomto zásobníku budou ukládat pouze jako číslo stavu) a kromě celkového počtu uložených hodnot Count také hodnotu Allocated. Jelikož se zásobník může zvětšit do velkých rozměrů, budeme muset při zaplnění celé paměti znovu alokovat další paměť, abychom mohli pokračovat ve vkládání. Alokace by měla probíhat exponenciálně, to znamená, že pokud bude hodnota Allocated rovna hodnotě Count, realokujeme paměť na její dvojnásobnou velikost (proměnnou Allocated pochopitelně také dvakrát zvětšíme). Čím více bude tedy velikost zásobníku růst, tím méně bude potřeba provádět realokací.

6.4.2 Struktura datových souborů

Abychom mohli ukládat nebo načítat automaty ze souboru, bude potřeba nevrhnout strukturu uložení automatů do souborů. Tyto struktury budou podobné jako struktury DFA a NFA popsané v předchozí kapitole.

Začněme strukturou uložení NFA. První hodnota (4 byty) reprezentuje počet stavů NFA.Q, druhá hodnota (opět 4 byty) reprezentuje počet symbolů abecedy NFA.E. Jako další následuje pole NFA.deltaPos, které má velikost Q×E+1. Potom následuje přechodová tabulka NFA.delta. Její velikost je určena posledním prvkem v poli deltaPos. Dále následuje jedna 4 bytová hodnota NFA.q0 reprezentující číslo startovního symbolu. Jako poslední je bitové pole reprezentující množinu koncových stavů, jeho velikost je dána počtem stavů. Každý stav reprezentuje jeden bit. Tato hodnota je nejbližším vyšším násobkem 4 bytů. Následující obrázek vše ještě jednou názorně zopakuje.



Obrázek 21 - Struktura souboru NFA

Pokračujeme strukturou uložení DFA. První hodnota (4 byty) reprezentuje počet stavů DFA.Q, druhá hodnota (opět 4 byty) reprezentuje počet symbolů abecedy DFA.E. Potom následuje přechodová tabulka DFA.delta. Její velikost je dána předchozími dvěma hodnotami Q×E. Dále následuje jedna 4 bytová hodnota DFA.q0 reprezentující číslo startovního symbolu. Jako poslední je bitové pole reprezentující množinu koncových stavů, jeho velikost je dána počtem stavů. Každý stav reprezentuje jeden bit. Tato hodnota je nejbližším vyšším násobkem 4 bytů. Následující obrázek vše ještě jednou názorně zopakuje.

Obrázek 22 - Struktura souboru DFA

Pro ukládání těchto struktur do souboru jsem napsal dvě funkce **WriteNFA(int,NFA)** a **WriteDFA(int,DFA)**. Podobně i pro načítání automatů jsem napsal funkce **NFA ReadNFA(int fd)** a **DFA ReadDFA(int fd)**. důležitým parametrem všech těchto funkcí je proměnná typu int, která obsahuje deskriptor otevřeného souboru z/do něhož se bude číst/zapisovat.

6.4.3 Funkce DFA SeqDetermination_1(NFA FA)

První funkce pro práci s konečnými automaty, kterou jsem implementoval v knihovně, je:

DFA SeqDetermination_1(NFA FA)

která v jediném parametru přijímá strukturu NFA a vrací DFA. Je to sekvenční verze algoritmu pro determinizaci NKA (function SeqDetermination(M)). Sekvenční verzi algoritmu implementuji především proto, abych mohl porovnávat zrychlení paralelních verzí se sekvenční. Dále budu zkoumat efektivitu paralelního řešení. Sekvenční řešení je vždy základ pro další zkoumání.

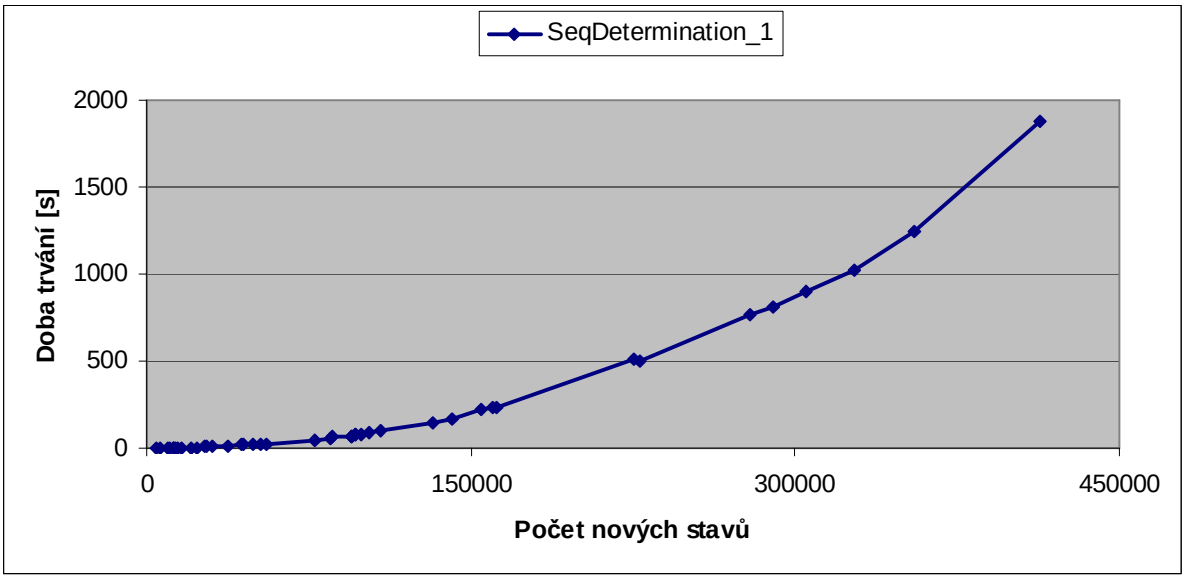
Při programování tohoto algoritmu bylo potřeba vyřešit několik zásadních implementačních otázek. Především to, jakým způsobem se bude pracovat s proměnnou LIST, která má sloužit jako tabulka množin stavů, která nám má převádět množinu stavů NKA na jeden příslušný stav DKA. Nejlepší by byla pochopitelně nějaká asociativní paměť, kterou bychom adresovali množinou stavů, a které by nám vracela číslo příslušného stavu. Takovou paměť však v počítači nemáme, takže nám bude muset postačit nějaké jednorozměrné pole. Budeme potřebovat pole množin typu t_set. Nebudeme muset ale ukládat číslo stavu nového automatu, protože toto číslo bude vždy rovno indexu v tabulce. V sekvenční verzi algoritmu se bude pole LIST procházet sekvenčně a bude se hledat množina a k ní se přečte odpovídající index. Bude se tedy procházet pole vždy od začátku. Tento způsob může mít jednu výhodu v případě, že bude ve vstupním automatu spousta nedefinovaných přechodů. Jako první množina stavů zde totiž bude uložena prázdná množina s číslem 0, která se najde hned při prvním porovnání. Aby se nemusely porovnávací funkcí porovnávat všechny množiny, tak se nejdříve porovná velikost množiny (položka Count) a v případě, že se velikost hledané množiny shoduje s velikostí právě zkoumané množiny, teprve potom se obě množiny porovnají. Aby bylo porovnání rychlé, budou v poli t_set (položka States) uloženy čísla stavů v seřazené posloupnosti. Takto nám stačí pouze porovnat dvě pole na rovnost a nemusíme se zajímat o to, že nám to pole vlastně reprezentuje nějakou množinu. Pokud se budou porovnávat dvě stejné množiny, tak všechna čísla stavů (pokud jsou seřazená) budou v poli uložena na stejných pozicích. Pokud by tomu tak nebylo, pak tyto množiny nejsou shodné.

Práce se seznamem LIST bude se zvyšujícím se počtem nových stavů stále pomalejší, protože se bude muset vyhledávat pořád ve větším a větším poli a to sekvenčně To bude problémem, když by nám mělo vzniknout velké množství stavů DKA.

Časová složitost této funkce je dána počtem nově vzniklých stavů, kterých může narůst až exponenciálně. Tomu se bohužel při determinizaci nevhodných automatů nemůžeme vyhnout. Naštěstí však většina automatů nikdy nedosáhne maximálního počtu stavů, většinou se jedná o výpočetně zvládnutelné hodnoty.

Výstupem tohoto algoritmu je tedy DKA, který je bez nedosažitelných stavů a navíc je úplně definovaný. Takže ho dále můžeme přímo použít už i k minimalizaci.

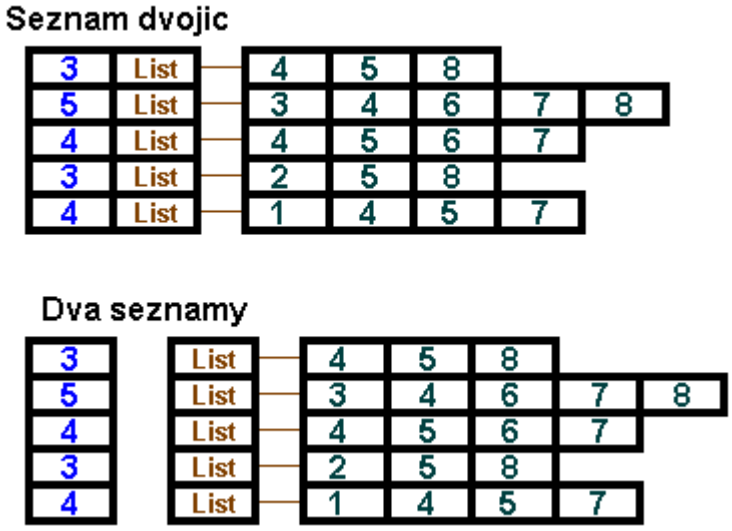
Na následujícím grafu je vidět časová závislost funkce SeqDetermination_1 na počtu vygenerovaných stavů. Čím větší počet stavů bude nový automat mít, tím déle bude trvat vyhledávání v seznamu (řádek 13). Graf má proto exponenciální tvar.



Od tohoto měření se budou odvíjet další výsledky. Budeme zkoumat možnosti dalšího paralelního zrychlení.

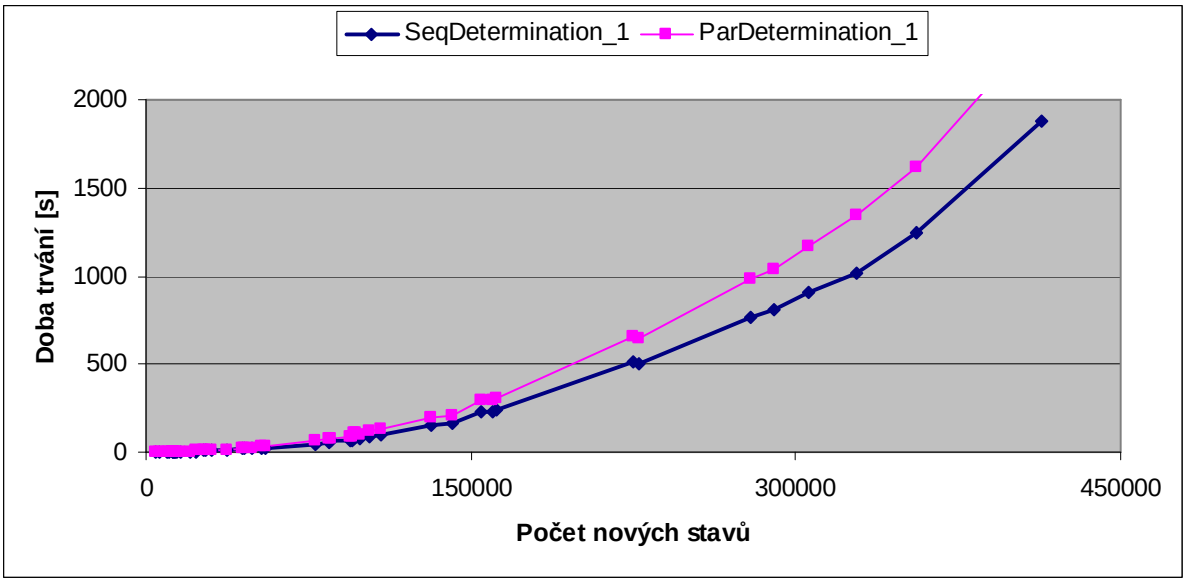
6.4.4 Funkce DFA ParDetermination_1(NFA FA)

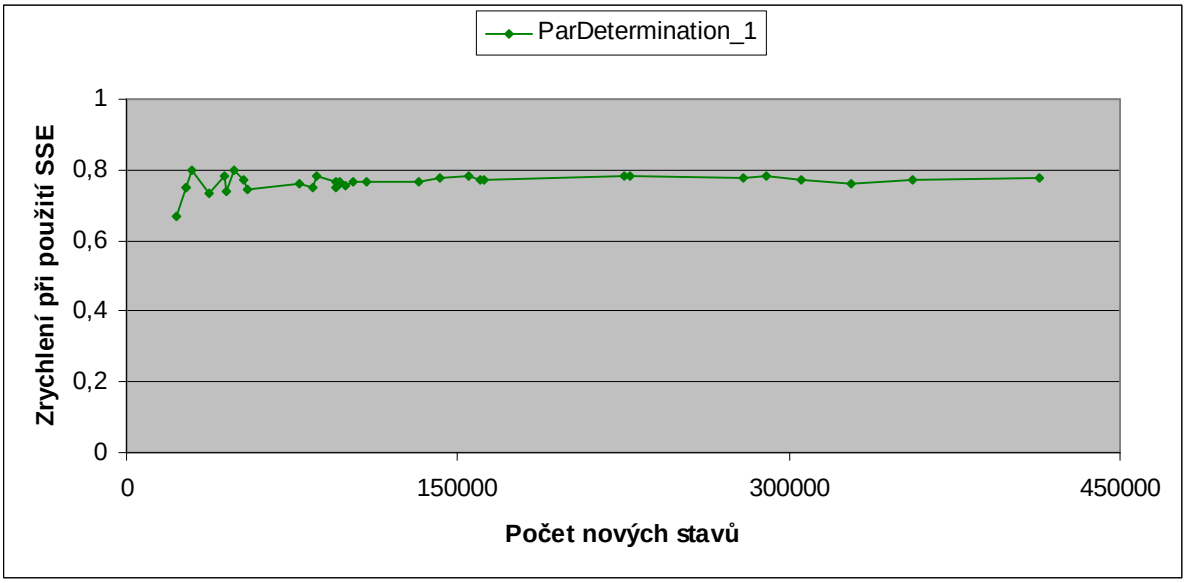
První věcí, kterou stojí za to vyzkoušet je, použit SSE instrukce při vyhledávání v seznamu LIST. Při porovnávání dvou množin nejprve porovnáme velikosti obou množit a teprve v případě shody porovnáváme i obsah seřazené posloupnosti stavů, které tyto množiny obsahují. Abychom mohli rychleji vyhledávat, upravíme strukturu LIST v programu. Tato struktura měla v předchozím algoritmu tvar dvojic (velikost množiny, seznam stavů množiny). Pomocí SSE budeme chtít procházet a porovnávat velikosti množin separátně v řadě. Nebylo by moudré tedy ukládat v paměti jeden seznam dvojic, nýbrž budeme ukládat dva. Jeden seznam bude seznam velikostí množiny a druhý bude seznam množin stavů. Tyto dvě pole (seznamy) budou v paměti uloženy odděleně, abychom je mohli rychleji procházet sekvenčně. Ilustrace je na obrázku 23. Pomocí SSE budeme procházet vždy 4 hodnoty paralelně a budeme porovnávat s hledanou hodnotou. Problém však je, že v poli bude existovat více stejných hledaných hodnot. Pokud tedy pomocí SSE nalezneme shodu, budeme muset zjistit, na kterém místě ze čtyř hodnot ke shodě došlo(může to být i na více místech současně). Pokud tedy SSL vrátí shodu, nezbude mi nic jiného, než všechny 4 hodnoty porovnat znovu sekvenčně a teprve tak zjistím, kde ke shodě došlo. Potom teprve (stejně jako v předchozím algoritmu) provedeme porovnání obou množin (seřazených posloupností). Toto porovnání se už nebude provádět pomocí SSE, ale použiji optimalizovanou standardní funkci memcmp(), která vyšla v předchozích testech jako nejoptimálnější řešení.



Obrázek 23 - Reprezentace proměnné LIST v paměti

Dále se tedy podíváme, jakých výsledků dosáhneme při použití funkce ParDetermination_1. Prováděl jsem měření na stejných automatech jako při sekvenční verzi, abych mohl porovnat případné zrychlení. V tomto případě bohužel k žádnému zrychlení nedošlo, protože použití SSE nebylo zrovna pro tento případ příliš vhodné. Naopak došlo ke zpomalení celého výpočtu. Jak je vidět na následujících dvou grafech, tak doba trvání determinizace byla ještě o něco větší než při použití sekvenční verze. Z druhého grafu si můžeme přečíst, že takto dosáhneme zrychlení kolem 0,8 (což je vlastně zpomalení výpočtu asi o 20%).





Proč v tomto případě nedošlo ke zrychlení? Pomocí SSE procházíme první pole (seznam velikostí množin). Všechny hodnoty se porovnávají s jednou hledanou hodnotou. Problém je v tom, že hledaná hodnota je nalezena v poli několikrát a vždy se příslušné čtyři položky prohledávají znovu sekvenčně. Pokud by hledaná hodnota byla v poli pouze jednou, pak by jistě k nějakému zrychlení došlo (podle testů provedených v kapitole 6.3.3), ale takto se provádí velké množství zbytečného porovnávání. SIMD instrukce se v tomto navrženém algoritmu nehodí.

Jiný způsob, jak využít SSE instrukce při determinizaci jsem nenalezl. Prvním neúspěšným pokusem se nenechám odradit. Proto v dalším algoritmu budu hledat možné zrychlení někde jinde. V další kapitole popíšu paralelní algoritmus, při kterém budu používat knihovnu OpenMP, se kterou budu programovat paralelní zpracování algoritmu na více procesorech (systémech).

6.4.5 Funkce DFA ParDetermination_2(NFA FA)

Nyní přejdeme k paralelnímu výpočtu na více procesorech, k tomuto účelu bude potřeba vhodně upravit algoritmus. Inicializace algoritmu bude stejná jako v předchozím případě, pouze hlavní část výpočtu doplníme o paralelní kód. Bude se jednat o téměř identickou funkci, však s tím rozdílem, že ji bude možno spustit na více procesorech současně a urychlit tak celkový výpočet. Na základě zkušeností v předchozí kapitole nebudu do programu znovu zavádět SSE instrukce, zaměřím se na zcela odlišný přístup k paralelizaci.

Dále zde uvádím modifikovaný algoritmus, který by již měl přinést zrychlení. Uvidíme dále, jakého zrychlení dosáhneme.

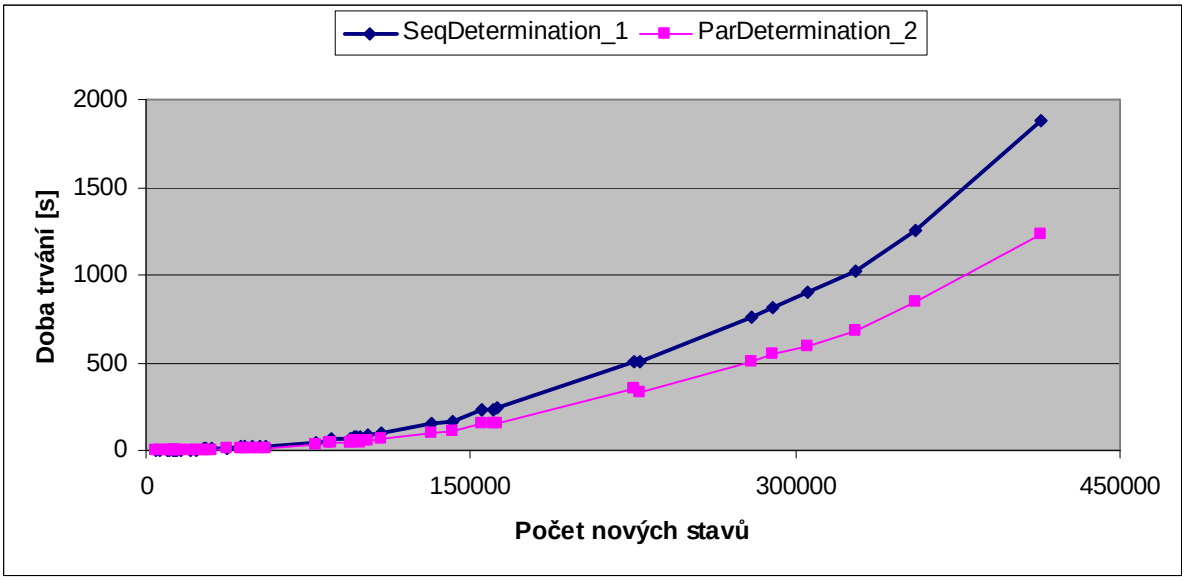
function ParDetermination_2(M):

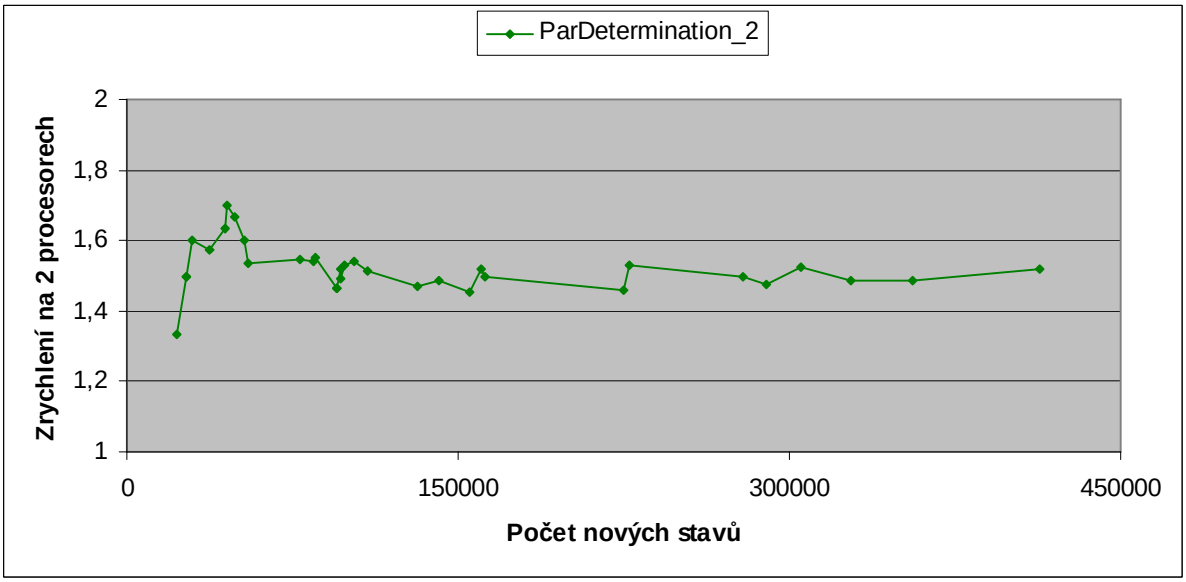
Inicializace stejná jako v předchozím algoritmu...

```
7.      while not STACK.EMPTY do
8.          q = STACK.POP()
9.          for a = 0 to |Σ| - 1 do in parallel
10.             tmp = ∅
11.             for each qq in LIST[q] do
12.                 tmp = tmp ∪ δ(qq, a)
13.             x = LIST.Find(tmp)
14.             if not found then
15.                 x = LIST.ADD(tmp) // vložíme nový stav a získáme jeho číslo
16.                 STACK.PUSH(x)
17.                 Q' = Q' ∪ {x}
18.                 if tmp ∩ F ≠ ∅ then
19.                     F' = F' ∪ {q}
20.             δ'(q, a) = x
21.     return M'
```

Oproti sekvenčnímu algoritmu tato verze obsahuje změnu na 9. řádku. Takto se budeme snažit provádět výpočet následujícího stavu pro více symbolů paralelně. Dále je nutné si uvědomit, že když pracujeme na více procesorech (nebo ve více vláknech programu), je potřeba zkontrolovat, zda by dvě vlákna nemohla ve stejný okamžik přistupovat ke stejným zdrojům. Řádky číslo 15 a 16 jsou v kódu vyznačen červenou barvou, protože vkládání do seznamu LIST by bez správného ošetření nemohla fungovat, protože by se mohlo stát, že by více vláken chtělo zapisovat současně do tohoto seznamu. Stejný problém by mohl nastat při zápisu do zásobníku. Takže prakticky celá větev podmíněného cyklu (řádky 15-19) je nebezpečným úsekem programu. Celý tento kód budeme muset umístit do kritické sekce, která bude zajišťovat sekvenční přístup k ní. Jinde kritická sekce být nemusí, v jiných částech algoritmu se buď pouze čtou hodnoty z paměti (což může provádět i více vláken současně) nebo se zapisuje na předem dané místo v paměti, na které zapisuje svůj výsledek vždy právě jedno vlákno (např. řádek 20 – zápis hodnoty přechodové funkce pro daný stav a konkrétní symbol). Datové struktury mohou zůstat stejné jako v předchozí verzi.

Práce s openMP je také jednoduchá, do kódu nám stačí připsat direktivy pro paralelní práci a pro kritickou sekci. pouze při překladu musíme uvést parametr **–fopenmp**, jinak by se algoritmus přeložil pouze jako sekvenční, všechen openMP paralelní kód by se ignoroval. Následuje test zrychlení. Testy byly prováděny na stejných automatech jako v sekvenční verzi. Podívejme se nyní na graf rychlosti a zrychlení a porovnejme výsledky se sekvenční verzí.





V grafu je vidět, že toto paralelní řešení skutečně přináší zrychlení výpočtu. K výpočtu jsem měl k dispozici pouze dvoujádrový procesor řady Intel Core 2 Duo. Z druhého grafu lze přečíst, jakého zrychlení bylo dosaženo. Zrychlení v této verzi se pohybovalo kolem 1,5, což je 50% zrychlení. Sice jsem použil dva procesory, ale ke dvounásobném zrychlení nedošlo, protože ta režie spojená s vytvářením vláken je docela veliká. Navíc byla paralelizována část programu uvnitř cyklu while, takže při každém průchodu cyklem while (nastane tolikrát, kolik bude všech nových stavů) nám tato režie pozdrží výpočet. Nejlepším řešením by bylo, pokusit se paralelizovat celý vnější cyklus while, tím by se na začátku algoritmu vytvořil tým vláken a režie by v porovnání s celým výpočtem byla zanedbatelná. Jak však paralelizovat cyklus while, ve kterém se vybírají hodnoty ze zásobníku? O řešení tohoto problému se čtenář dozví v následující kapitole.

6.4.6 Funkce DFA ParDetermination_3(NFA FA)

V této kapitole si představíme poslední verzi paralelního algoritmu pro determinizaci NKA. Budeme chtít paralelizovat celý vnější cyklus while a zajistit tak menší režii spojenou s vytvářením vláken v openMP. Budeme proto ještě více muset upravit algoritmus, protože každé vlákno bude potřebovat vstupovat do zásobníku, vybírat z něj hodnoty a také vkládat nové hodnoty. To je samozřejmě problém. Vlákna nesmí ve stejný okamžik přistupovat do stejného místa v paměti a zapisovat. Proto předcházející algoritmus ještě vylepším, aby k tomuto nemohlo dojít a aby program pracoval správně. Úprava byla provedena následujícím způsobem:

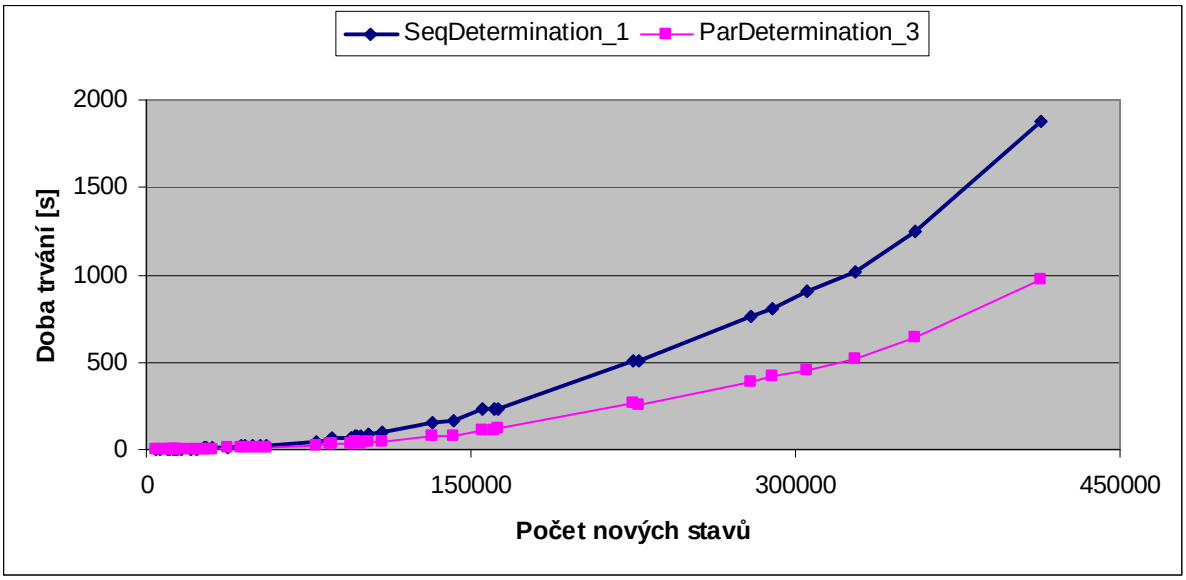
function ParDetermination_3(M):

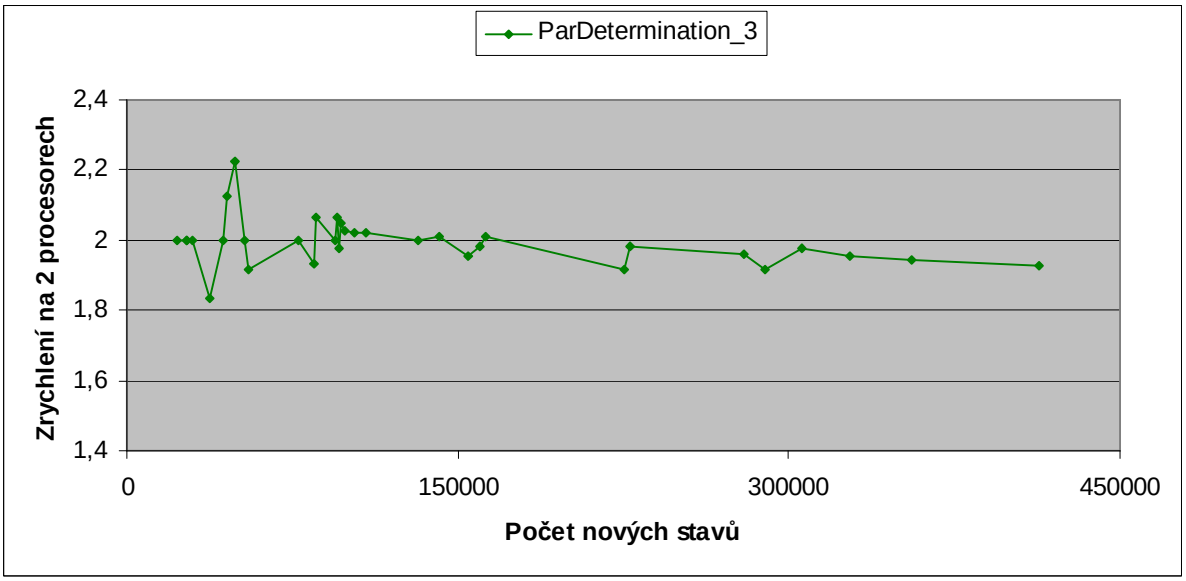
1. Buďte proměnné STACK1 a STACK2 nové inicializované seznamy
2. Buď proměnná LIST nový inicializovaný číslovaný indexovatelný seznam
3. Buď $M' = (Q', \Sigma', \delta', q_0', F')$ nový DKA
4. Q' = {0,1}; $\Sigma' = \Sigma$; $q_0' = 1$; $F' = \varnothing$
5. LIST.ADD($\varnothing$); LIST.ADD({ q_0 })
6. STACK1.ADD(q_0')
7. **while not** STACK1.EMPTY **do**
8. **for** q = 0 **to** |STACK1| - 1 **do in parallel**
9. **for** a = 0 **to** $|\Sigma|$ - 1 **do**
10. tmp = $\varnothing$
11. ***for each*** *qq* ***in*** LIST[q] **do**
12. $tmp = tmp \cup \delta(qq, a)$
13. x = LIST.Find(tmp)
14. **if not** found **then**
15. x = LIST.ADD(tmp) // vložíme nový stav a získáme číslo
16. STACK2.ADD(x)
17. $Q' = Q' \cup \{x\}$
18. **if** $tmp \cap F \neq \varnothing$ **then**
19. $F' = F' \cup \{q\}$
20. $\delta'(q, a)$ = x
21. **swap** (STACK1, STACK2)
22. **return** M'

Pro tento případ použijeme dva zásobníky. V prvním z nich se budou nacházet položky určené k vybírání. Bude se vlastně jednat pouze o pole hodnot, které se paralelně rozdělí na zpracování více vláknům. Nebude vadit, že to nebude zásobník, protože nám nezáleží na pořadí, ve kterém jsou dané stavy ze zásobníku vybírány. Druhý zásobník bude sloužit pouze pro vkládání hodnot. Po vyprázdnění prvního zásobníku se prohodí role obou zásobníků a algoritmus bude pokračovat dál. Algoritmus by šel i navrhnout s jednou frontou, do které by vlákna zapisovala i četla, ale tato fronta by musela být uzavřena ve speciální kritické sekci, do které by se vstupovalo docela často. Použitím dvou front se kritické sekci můžeme vyhnout zcela a ušetřit tak drahocenný čas pro užitečnou činnost programu.

Jedné kritické sekci se nevyhneme ani v tomto případě. Pokud budeme do zásobníku nebo fronty přidávat hodnoty, musíme to udělat stejně jako v předchozím případě, jinak by došlo k nekonzistenci hodnot a nesprávnému výpočtu. Proto řádky 15-19 musí být chráněny kritickou sekcí.

Podívejme se nyní na výsledky testů, kterých jsme dosáhli při měření rychlosti výpočtu.





Přiznám se, že výsledky testů mě velmi překvapily. Zdá se, že tento způsob byl nejvhodnější. Z grafů můžeme vyčíst, že zrychlení algoritmu na dvou procesorech byl kolem 2. Dosáhli jsme dvojnásobného zrychlení, to znamená, že efektivita výpočtu byla téměř 100%. Režie dvou zásobníků, kterou jsme přidali do programu se ukázala být bezvýznamnou. Také jsme výrazně snížili režii vytváření vláken v openMP, takže můžeme celou režii zanedbat a prohlásit, že takto napsaný algoritmus je optimální pro víceprocesorové výpočetní stroje. Použití openMP výhradně doporučuji tam, kde můžeme paralelizovat vnější cykly. V mém případě se výsledky potvrdily.

Tímto končí kapitola zabývající se determinizací NKA. Ukázali jsme si tři způsoby, jakými jsem provedl paralelizaci sekvenčního algoritmu. Ať už s dobrými nebo špatnými výsledky, vždy jsem měřil a testoval na počítači s procesorem Intel Core 2 Duo, (dvoujádrový procesor) 2048 MB RAM (taková velikost paměti však nebyla vůbec potřeba, vždy se všechny výpočty vešly do paměti, nikdy nebylo potřeba odkládat mezivýsledky na disk). Struktury pro reprezentaci DKA a NKA nebyly nikdy tak velké, aby se nevešly do paměti. Většinou zabíraly několik KB a práce s nimi byla rychlá. Problém byl především s převodní tabulkou LIST, která sloužila pro převod množiny stavů (NKA) na jeden příslušný stav nového NKA. tato tabulka po celou dobu výpočtu stále rostla a ztěžovala celý výpočet, protože jsme museli vyhledávat pořád ve větším a větším poli. Jedním nápadem, jak toto pole reprezentovat, bylo pomocí stromu. Tento způsob jsem ale zavrhnul hned na počátku, protože každá položka by musela obsahovat tolik ukazatelů, kolik bychom měli stavů. takže jedna jediná hodnota by mohla v paměti zabírat i několik KB. Když bychom měli v paměti třeba 1000000 stavů, potom by tato tabulka měla i několik GB. Časová složitost průchodu stromem by byla sice logaritmická, avšak za cenu neúnosně velké paměťové náročnosti.

Dále nám zbývá věnovat se minimalizaci DKA.

6.5 Minimalizace DKA

Když už nyní máme úspěšně determinizovaný náš vstupní automat, nyní bude potřeba provést poslední fázi – redukovat počet jeho stavů. Budu zde prezentovat dva algoritmy, každý z nich implementuji jak v sekvenční tak i v paralelní verzi, abychom mohli vidět zrychlení, pokud nějakého dosáhneme. Minimalizace už nebude tak náročná jako determinizace.

Všechny datové typy, které se používaly v předchozím algoritmu budeme moci použít i zde. Především využijeme datového typu DFA. Protože algoritmus minimalizace bere jeden parametr DFA a vrací také DFA.

6.5.1 Minimalizace_1

Následující algoritmus vychází z [Min1].

1. Stavy jsou rozděleny na dvě množiny, koncové stavy a nekoncové stavy. To proto, že koncové stavy umí akceptovat prázdný řetězec, nekoncové stavy nikoliv, jsou rozlišitelné. Toto bude základní rozdělení $Q'' = \{ F, Q\text{-}F \}$.
2. Předpokládejme k-té rozdělení, budeme mít m množin, $Q'' = \{ I_1, I_2, ..., I_m \}$. Pro všechna I_i a každé $a \in \sum$, provedeme přechod v automatu. Množina I_i bude rozdělena na menší podmnožiny $I_{i1}, I_{i2},...I_{ij}$, podle provedených přechodů.
3. Krok 2 opakujeme, dokud vznikají další podmnožiny. Množina, kterou nakonec dostaneme, nazveme Q'.
4. Pro každou podmnožinu v Q', pokud obsahuje originální startovní symbol, bude tato podmnožina reprezentovat startovní symbol v novém automatu. Stejně tak s koncovými stavy. Pokud obsahuje nějaký koncový stav, bude výsledný stav také koncovým stavem.
5. Odstranění nedosažitelných stavů je posledním krokem.

6.5.1.1 Tabulka rozlišitelných stavů

Tabulka rozlišitelných stavů bude potřeba v následujícím algoritmu. První řádek tabulky se stavem q0 je vynechán stejně jako poslední sloupec, protože reprezentace stavů (p,q) je stejná jako (q,p). Opakující se stejné stavy jsou označeny „–„ všechny ostatní položky jsou inicializovány na null. Konstrukce tabulky rozlišitelných stavů je ukázána v následující tabulce.

q_1		-	-	-	-
q_2			-	-	-
q_3				-	-
q_4					-
q_5					
	q_0	q_1	q_2	q_3	q_4

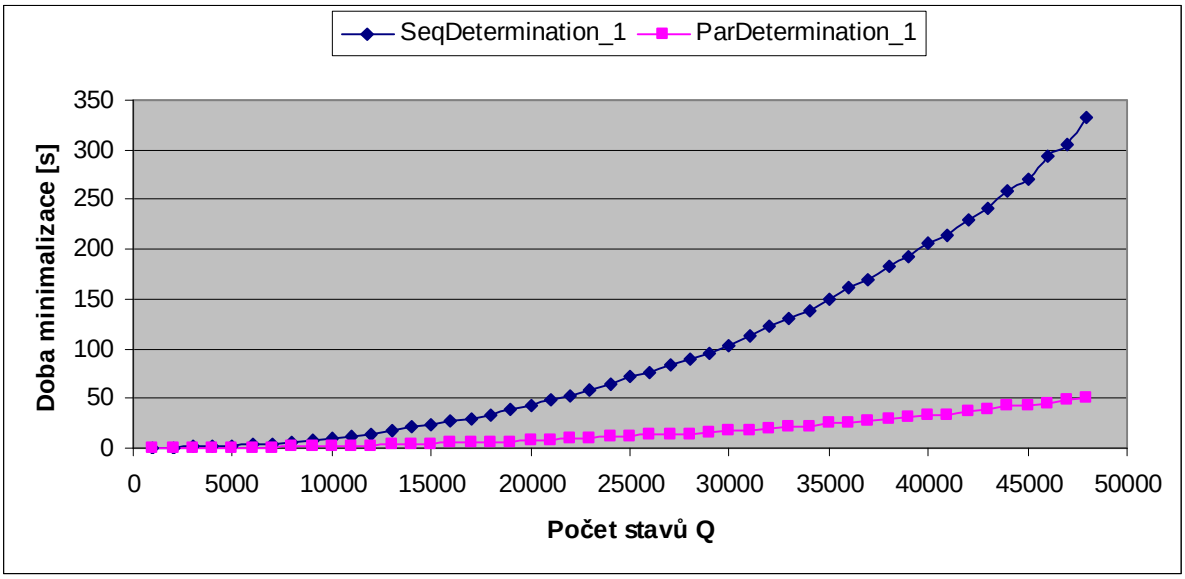
6.5.1.2 Paralelní zpracování pomocí tabulky rozlišitelných stavů

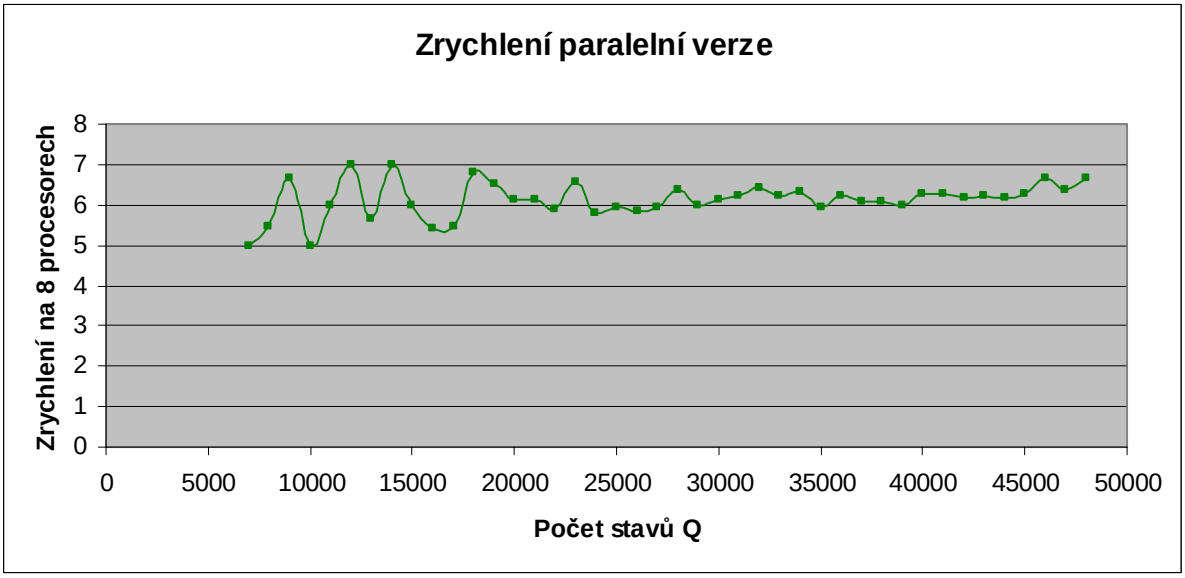
Analyzujeme tabulku rozlišitelných stavů. Nejdříve rozdělíme stavy podle koncových stavů. To lze provést paralelně. Potom další páry, které jsou spojeny přechodem s rozlišitelnými páry mohou být rozlišeny paralelně.

Algoritmus pro paralelní zpracování:

1. Sestrojíme tabulku rozlišitelných stavů, jak bylo ukázáno předchozí metodou.
2. **for each** (q,p), **where** $(q \in F, p \in (Q-F))$ **do in parallel**
3. Hodnoty (q,p) v tabulce jsou označeny symbolem „X“
4. **do**
5. **for each** (q,p) **where** $(q \in F, p \in F)$ **or** $(q \in (Q-F), p \in (Q-F))$ **and** $q \notin p$
   ---- **and** (q,p) není v tabulce označen **do in parallel**
6. **if** $\exists a \in \sum$, kde hodnota $(\delta(q,a), \delta(p,a))$ byla v tabulce označena „X“
   --------- nebo obsahuje pouze nedosažitelný stav **then**
7. (q,p) bude označena v tabulce „X“
8. **while** Tabulka rozlišitelných stavů se změnila
9. Nalezneme páry stavů (q,p), které nebyly v tabulce označeny, tyto stavy jsou nerozlišitelné. Takto z tabulky můžeme dostat minimální DKA.

Pokud budeme mít k dispozici $P=Q^2/2$ procesorů, potom paralelní cyklus na 5. řádku bude zpracován v konstantním čase O(1) a celkový čas bude záviset pouze na počtu provedení cyklu do-while, což může být v nejhorším případě Q^2. V praxi to však nebývá většinou ani 10 kroků, takže teoreticky bychom mohli vzít tuto hodnotu jako malou konstantu. Dále je tu velká paměťová náročnost. musíme mít uloženou tabulku všech dvojic stavů, tedy paměťová náročnost takové tabulky je Q^2. To je jeden ze zásadních problémů, protože takto můžeme úspěšně otestovat pouze automaty s menším počtem stavů, v praxi to byly maximálně desítky tisíc stavů. Podívejme se nyní na graf závislosti doby zpracování na počtu stavů. Test jsem provedl na školním serveru merlin.fit.vutbr.cz.





V grafech je dobře vidět, že obě verze, sekvenční i paralelní, mají exponenciální tvar grafu. Paralelní verze však vykazuje velké zrychlení. Na osmi procesorech se zrychlení algoritmu pohybuje kolem 6-7. Tento algoritmus je tedy dobře paralelizovatelný, avšak za cenu velké paměťové náročnosti.

Funkce jsou k dispozici v knihovně, deklarované v hlavičkovém souboru FA.h. Jedná se o tyto dvě funkce:

DFA SeqMinimizeDFA_1(DFA FA);

DFA ParMinimizeDFA_1(DFA FA);

Jejich parametrem je struktura DFA popsaná v předcházející kapitole.

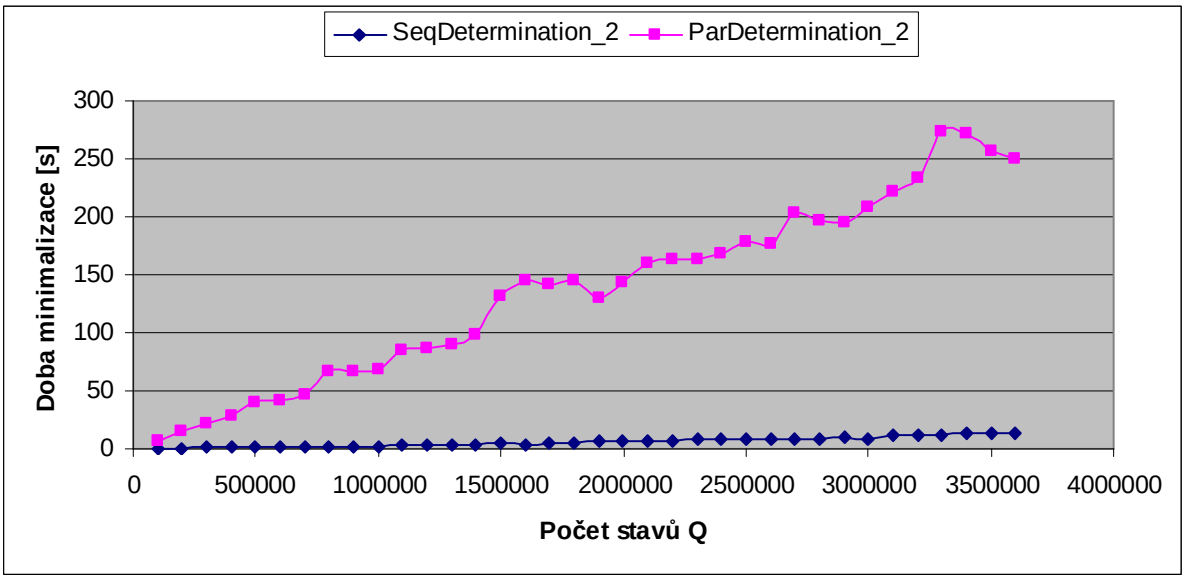
6.5.2 Minimalizace_2

Poslední verze minimalizačního algoritmu vychází z [Min2].

1. Buď BLOCK tabulka. Pro každý stav obsahuje informaci, do jakého bloku patří.
2. Buď LABEL tabulka. Pro každý stav obsahuje popisek (dvojici bloků)
3. **do**
4. **for each** a **in** Σ **do**
5. **for each** q **in** Q **do in parallel**
6. B = BLOCK$[q]$
7. $B' = \text{BLOCK}[\delta(q,a)]$
8. LABEL[q] = (B, B')
9. Seřaď tabulku LABEL a rozděl do tabulky BLOCK všechny stavy tak, aby
... stavy se stejným popiskem byly ve stejném bloku.
10. **while** vznikají další bloky

Pro tento algoritmus potřebujeme dvě pomocné tabulky, BLOCK a LABEL, obě mají paměťovou náročnost lineárně závislou na počtu stavů. Oproti předchozímu algoritmu je to velké zlepšení. Vnější smyčka (řádek 3) je sekvenční a zdá se být těžce paralelizovatelná. Ale experimenty ukazují, že počet cyklů této smyčky je velmi malý, většinou menší než 10.

Při testování tohoto algoritmu jsem zjistil, že nejnáročnější operací celého algoritmu je řazení na řádku 9. V sekvenční verzi tedy použiji řadicí algoritmus QuickSort, který je optimálním sekvenčním řadicím algoritmem. Testy dále ukázaly překvapivou rychlost sekvenční verze. Zatímco sekvenční verze trvá řádově sekundy, paralelní verze s použitím openMP trvá několik minut. openMP se tedy ukázalo jako nevhodné pro použití s tímto algoritmem. Jak je vidět na následujícím grafu.



6.6 Praktické využití knihovny

Pro testování knihovních funkcí jsem napsal demo program FA, který podle zadaných přepínačů a parametrů volá požadované funkce. Je napsán v souboru main.c. a může být přeložen a spuštěn na libovolném počítači s překladačem gcc podporujícím SSE a openMP. Překlad lze provést příkazem make. Po přeložení vznikne spustitelný soubor FA, který realizuje práci s knihovnou.

Parametry programu můžou být následující:

--determine N	Zadaný automat se bude determinizovat pomocí algoritmu číslo N.
--minimize N	Zadaný automat se bude minimalizovat pomocí algoritmu číslo N.
--par	Místo sekvenčního algoritmu se spustí patřičný paralelní algoritmus.
--NFA -Q X -E Y	Vygeneruje se náhodný NKA s X stavy a Y symboly abecedy.
--DFA -Q X -E Y	Vygeneruje se náhodný DKA s X stavy a Y symboly abecedy.
--file filename	Náhodně vygenerovaný automat bude před zpracováním uložen do souboru.

Při standardním použití se vstupní automat očekává na standardním vstupu programu. Pouze pokud je zadaný přepínač --NFA nebo --DFA, bude vstupní automat vygenerován náhodně pomocí funkce RandomNFA nebo RandomDFA. V obou případech musí být také zadány parametry generovaného automatu pomocí přepínačů -Q a -E. Pokud navíc zadáme přepínač --file, můžeme si vygenerovaný automat ještě před samotným zpracováním uložit do souboru pro pozdější znovupoužití. Zavolá se funkce WriteNFA nebo WriteDFA Takto si můžeme vygenerovat jeden testovací automat a na něm spustit všechny možné algoritmy a porovnat jejich rychlosti.

Výstupní automat je vždy zapisován na standardní výstup. Uživatel si tedy může sám rozhodnout, zda výstup přesměruje do souboru nebo ho může přesměrovat na vstup další instance programu. Takto můžeme provést zřetězení operací. Např. jedna instance programu může zadaný NKA determinizovat, její výstup může být přesměrován na standardní vstup druhé instance, která provede minimalizaci. Příklad takového zápisu je zde:

```
./FA --determine 3 --par --NFA -Q 100 -E 8 --file automata.nfa | ./FA --minimize 2 > output.dfa
```

Tento zápis znamená, že nejprve bude první instancí programu vygenerován náhodný NKA, tento bude uložen do souboru automata.nfa. Následuje paralelní determinizace algoritmem ParDetermination_3, determinizovaný automat bude přesměrován druhé instanci, která provede minimalizaci sekvenčním algoritmem SeqMinimize_2. Výstupní minimální automat bude nakonec přesměrován a uložen v souboru output.dfa.

Tento demo program jsem používal při testování algoritmů a jeho využití může být i v praxi při práci s konečnými automaty na standardním vstupu programu. Stačí, když bude vstupní struktura automatu ve správném tvaru (tak, jak je popsána v kapitole 6.4.2).

7 Závěr

V této práci jsem se zabýval dvěmi základními operacemi nad konečnými automaty. Determinizací NKA a minimalizací DKA.V obou případech jsem se snažil navrhnout optimální algoritmus tak, aby se dal co nejlépe paralelně řešit na více procesorech současně. Neskončil jsem pouze u jednoho algoritmu, vždy jsem navrhnul více verzí a poté testoval a porovnával, jejich rychlost Všechny paralelní algoritmy jsem srovnával s jejich sekvenčními verzemi.

Při hledání zrychlení jsem pro paralelní determinizaci vytvořil tři algoritmy. První algoritmus používal SSE instrukce a ukázal se pro tento případ nevhodný. Zjistil jsem např., že standardní funkce jazyka C (jako je memcpy nebo memcmp) jsou už optimalizované natolik dobře, že už ani pomocí SSE jsem nedokázal urychlit jejich běh. Použití SSE instrukcí se vyplatí především tam, kde je potřeba provádět nějaký výpočet nad vektory čísel. Pro takové operace neexistují standardní funkce, programátor si je musí sám vytvořit. Potom mnohem lepších výsledků dosáhneme, pokud použijeme SSE místo jednoduchého cyklu.

Ve druhé verzi paralelního algoritmu jsem dosáhl mnohem lepších výsledků při použití knihovny openMP, pomocí níž jsem programoval paralelní části kódu. Úspěšně se mi podařilo paralelizoval vnitřní cyklus algoritmu a dosáhnout tak zrychlení oproti sekvenční verzi. Ještě to ale nebylo zcela optimální, účinnost byla kolem 75%. To bylo dáno velkou režií, kterou potřebuje openMP při vytváření vláken a inicializaci paralelních sekcí. Bylo potřeba tuto režii co nejvíce minimalizovat, aby se neprojevila tak výrazně jako v druhém algoritmu.

Proto vznikla třetí verze, která dokáže paralelizovat celý vnější cyklus. Bylo sice potřeba mírně upravit celý algoritmus, použít dva zásobníky místo jednoho, avšak to snažení se nakonec vyplatilo. Tato poslední verze algoritmu dosáhla velmi výrazného zrychlení. Na dvou procesorovém počítači běžel výpočet dvakrát rychleji oproti sekvenční verzi. To znamená, že účinnost zde byla téměř 100%. V tomto případě byla režie spojená s inicializací paralelní sekce minimální.

Další operace, kterou jsem se snažil implementovat do knihovny byla minimalizace DKA. Zde jsem měl k dispozici dva různé sekvenční algoritmy a každý z nich jsem se snažil nějakým způsobem urychlit.

První algoritmus používal ke svému výpočtu dvourozměrnou tabulku rozlišitelných stavů, do které se v každém kroku zaznamenávalo, zda je daná dvojice stavů rozlišitelná, či ne. Jelikož se zde pracovalo ve velkém poli, a pro každou položku bylo v každém kroku potřeba vypočítat nějakou hodnotu, proto se tento algoritmus dal velmi dobře paralelizovat pomocí openMP. Účinnost paralelní verze se pohybovala kolem 90%. Avšak v praxi by byl tento algoritmus nepoužitelný, protože měl jak velkou paměťovou náročnost tak i velkou časovou složitost. Tímto algoritmem bychom mohli minimalizovat pouze takové automaty, které by neměly více než desítky tisíc stavů.

Zato druhý algoritmus je na tom opačně. Testy ukázaly, že jeho časová složitost je dána především kvalitou řadicího algoritmu, který se použije. Proto jsem v sekvenční verzi použil řadicí algoritmus QuickSort. Díky tomu trval sekvenční minimalizační algoritmus pouhé sekundy. Tento algoritmus velmi dobře zvládal automaty s několika miliony stavy. Paralelní verzi jsem se pokusil opět naprogramovat pomocí openMP, avšak paralelní cyklus zde byl příliš hluboko vnořený v jiných cyklech a smyčkách, že se opět velmi výrazně projevila režie. Tímto byl celý výpočet zdržován takovým způsobem, že došlo až k několikanásobnému zpomalení oproti sekvenční verzi.

Pro všechna testování jsem naprogramoval demo program. Tento program používá celou implementovanou knihovnu a můžeme pomocí něj volat funkce z této knihovny. Čtenář sám si může vyzkoušet a ověřit funkci všech algoritmů. V knihovně pochopitelně nechybí funkce pro porovnávání dvou automatů a samozřejmostí jsou i funkce pro ukládání a načítání automatů ze souboru. Za zmínku také ještě stojí funkce pro generování náhodných automatů.

Literatura

[1] Wikipedia: Konečný automat, aktualizováno 2009-08-04 Dostupné na URL: <http://cs.wikipedia.org/wiki/Kone%C4%8Dn%C3%BD_automat>

[2] Wikipedia: Finite-state machine, aktualizováno 2009-12-17 Dostupné na URL: <http://en.wikipedia.org/wiki/Finite_automata>

[3] M. Češka, T. Vojnar, A. Smrčka: Teoretická informatika TIN Studijní opora, aktualizováno 2009-11-18, Dostupné na URL: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/TIN-IT/texts/oporaTIN.pdf

[4] M. Češka, T. Vojnar: Teoretická informatika, aktualizováno 2009, Dostupné na URL: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/TIN-IT/lectures/prednasky/tin-pr01-rj1.pdf

[5] M. Češka, T. Vojnar: Teoretická informatika Minimalizace DKA, aktualizováno 2009, Dostupné na URL: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/TIN-IT/lectures/prednasky/tin-pr02-rj2.pdf

[6] Blaise Barney, Lawrence Livermore National Laboratory: Introduction to paralel computing, aktualizováno 2010-01-04, Dostupné na URL: https://computing.llnl.gov/tutorials/parallel_comp/

[7] Wikipedia: Paralel computing, aktualizováno 2009-11-04, Dostupné na URL: http://en.wikipedia.org/wiki/Parallel_computing

[8] Faith E. Fich: The komplexity of computation on the parallel random access machine, Dostupné na URL: https://www.fit.vutbr.cz/study/courses/PDA/private/Reif21_PRAMsurvey.pdf

[9] OpenMP: The OpenMP API specification for parallel programming, aktualizováno 2009-12-11, Dostupné na URL: http://openmp.org/wp/about-openmp/

[10] Jiří Furst: Úvod do OpenMP, aktualizováno 2009, dostupné na URL: http://www.civ.cvut.cz/ESF/info/vloz2.php?oid=10&oname=Uvod_do_OpenMP.pdf

[11] Paměti - Návrh počítačových systémů - INP, dostupné na URL: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/INP-IT/lectures/inp09_11pameti_v1.pdf

[12] Wikipedia: Computer architecture, aktualizováno 2010-01-03, Dostupné na URL: http://en.wikipedia.org/wiki/Computer_architecture

[13] David Martinek: Profilování a optimalizace programů, aktualizováno 2009-02-11, dostupné na URL: http://www.fit.vutbr.cz/~martinek/clang/profiling.html.cs

[14] GNU gprof, aktualizováno 2003-07, Dostupné na URL: http://www.delorie.com/gnu/docs/binutils/gprof_toc.html

[Min1] Yu-Qiang Sun, Hai-Lian Lu, Yu-Ping Li, Hai-Yan Wang: Parallel Processing of Minimization Algorithm for Determination Finite Automata

[Min2] B. Ravikumar, X. Xiong: A parallel algorithm for minimization of finite automata, Department of computer science, University of Rhode Island

Seznam příloh

Příloha 1. CD