KIV/VSS

1.9. – Generování náhodných čísel

Gaussovské rozdělení

Miroslav Liška – A17N0081P

topiker@students.zcu.cz

9.12.1992

28. listopadu 2017

1 Zadání

1.1 Teoretické pozadí

Zejména při fyzické aktivitě nebo příjmu potravy dochází k výrazné změně koncentrace glukózy. V zadaných datech máte několik měření koncentrací glukózy v intersticiální tekutině [mmol/l]. Jsou vzorkovaná po 5 minutách tzv. systémem CGMS. U každého měření je časová značka a měření jsou rozdělena do segmentů, které trvají od několika hodin do několika dní. Vaším úkolem je identifikovat cca 3 - 5 významných kolísání koncentrací glukózy během dne (uvažuje se 3 jídlo a 2 fyzická zátěž). Při identifikaci si můžete pomoci i časovými značkami, případně naměřenou koncentrací v krvi, která by po jídle a při fyzické aktivitě měla být taktéž zvýšená.

V běžném životě si pacient údaje o jídle a fyzické aktivitě zadává manuálně a ještě s chybou. Systém automatické detekce by tuto chybu redukoval a napomohl tak lepší léčbě pacienta. Práce tedy není "šuplíková", ale má praktické využití (cca každý 11. člověk má diabetes a polovina z nich o tom ani neví, protože diabetes nebolí, dokud není pozdě).

Změny koncentrace glukózy lze nejjednoduššeji detekovat jako ohodnocené plovoucí okno - tj. fixní časový úsek, který bude "plout"v čase segmentu od jeho počátku až na konec. Ohodnocení okna může být součet rozdílů koncentrací glukózy v daném okně. S algoritmem lze dále experimentovat, např. velikost okna a mezní ohodnocení (tj. od kdy je okno považováno za významnou změnu koncentrace glukózy) lze určovat např. pomocí Diferenciální evoluce, nebo jiným algoritmem - může to být i 2D půlení intervalu. Vlastní invenci při vývoji detekčních algoritmu se meze nekladou.

1.2 Verze úlohy

Zpracujte úlohu alespoň ve dvou verzích ze tří možných:

- Paralelní program pro systém se sdílenou pamětí
- x86 CPU + OpenCL/C++ AMP GPGPU
- Paralelní program pro systém s distribuovanou pamětí

1.3 Data

Naměřené hodnoty jsou uloženy ve formátu SQLite verze 3. Konkrétně jsou uloženy v tabulce measuredvalue. Požadované hodnoty najdete ve sloupci ist, který vyjadřuje koncetraci v intersticiální tekutině v [mmol/l]. Čas měření je zanesen ve sloupci measuredat, a je ve formátu ISO 8601. Data jsou seskupena do tzv. segmentů, viz sloupec segmentid. Naměřená data zpracovávejte vždy po celých segmentech. Jméno segmentu lze dohledat v tabulce timesegment a jméno pacienta analogicky v tabulce subject.

1.4 Výstup

Na stdout vypište získané statistické ukazatele jako tabulku v csv formátu. Zároveň vygenerujte grafický výstup ve formátu SVG (pro každý segment jedno SVG), ve kterém graficky znázorníte změny koncentrace glukózy považované za příjem potravy, fyzickou aktivitu, apod. Implentujte přepínač, který buď segment vykreslí v celé jeho délce, anebo ho bude zalamovat po 24 hodinách - tj. osa X (čas) bude mít hodnoty od 00:00 do 23:59. V takovém případě by mohlo být vidět, např. zda pacient snídán či večeří pravidelně - což je také možná nápověda pro detekční algoritmus.

1.5 Další statistiky

Program také spusťte s jedním vláknem/procesem a změřte čas výpočtu sériovým kódem a čas výpočtu paralelizovaným kódem (pro všechny verze paralelizovaného kódu zvlášť). Z těchto hodnot vypočítejte následující ukazatele:

- Amdahlův zákon, f – čas sériově prováděné části kódu
- Gustafsonův zákon, a – část kódu, kterou nelze paralelizovat
- Karp-Flattova metrika, e – část sériově prováděného kódu

2 Analýza

2.1 Detekce změn koncentrace glukózy

O chování koncentrace glukózy v krvi víme (informace z přednášek), že pokud pozorovaný subjekt zkonzumuje nějakou potravinu, konentrace vzroste. Pokud je subjektem vynaložena nějaká aktivita, koncentrace typicky mírně vzroste a pak začne klesat. Tyto akty pak v datech generují významné kolísání, jejichž detekce je cílem práce. Dále je z přednášky známo, že kolísání trvá typicky tři hodiny s tím, že nárůst trvá hodinu a následné klesání pak dvě hodiny.

Jedním z možných řešení detekce je nalezení lokálních extrémů. Následně se pro každý extrém vezmou spojitě data začínající před extrémem a po extrému o nějaké velikosti. Pro výběr nejlepších výsledků je nutné získané intervaly ohodnotit. Dále je potřeba nějakým způsobem naložit s překrývajícími se intervaly, například jejich sloučením či vyřazením horšího.

Dalším řešením může být evoluční genetický algoritmus. Data jsou rozdělena náhodně na intervaly o pevné velikosti a pro každý interval je vypočítána jeho fitness funkce. Následně se vybrané intervaly posunou a spočítá se jejich fitness funkce a tím vznikne nová generace intervalů. Podle chování genetického algoritmu se s novou generací patřičně naloží. Posouvání probíhá do té doby, dokud nevznikne nejlepší generace intervalů, které jsou pak detekovaným kolísáním.

2.1.1 Zvolené řešení

Pro detekci kolísání jsem se rozhodl použít algoritmus posuvného okénka. Princip spočívá v tom, že se napříč daty iteruje tzv. okénkem o velikosti n. Každá iterace posune okénko o jedno měření dál. Každé okénko je ohodnoceno funkcí. V implementaci je jako funkce zvoleno rozdíl sousedních hodnot na druhou. Na druhou z toho důvodu, aby nebylo okénko ohodnoceno záporně.

Jakmile je spočítáno ohodnocení všech okének, je potřeba vybrat pouze ty nejlepší a nějakým způsobem naložit s okénkami, které se překrývají. Při výběru významějších okének jsem se rozhodl vybrat pouze ty, jejichž ohodnocení je lepší, než průměrná hodnota. Po výběru významějších okének je možné, že se budou jednotlivé intervaly překrývat. Je předpokladem, že ve vybraná okénka budou reprezentovat části s významným klesáním či nárůstem. Pokud se tedy intervaly překrývají, je vhodné je spojit. Pro takto spojená okénka znovu spočítáme jejich ohodnocení a následně se vezme n nejlepších.

2.2 Načtení uložených dat

Naměřené hodnoty jsou ve formátu SQLite verze 3. Pro přístup k datům je tedy vhodné přistoupit za pomocí SQL dotazů. Při analýze bylo zjištěno, že ne všechny segmenty uvedené v tabulce segmentů mají i naměřená data a naopak nějaká naměřená data mají id segmentu, které není v tabulce segmentů. Při načtení dat z databáze budou tedy zvolena pouze ta data, jejichž id segmentu odpovídá tabulce segmentů. Pro práci hledání kolísání bude nutné, aby data

byla seřazena podle data měření (measuredate). Datum měření je uloženo ve formátu ISO 8601. Podle doporučení je dobré převést datum na formát, kdy hodnota 1.0 představuje datum 1.1.1900.

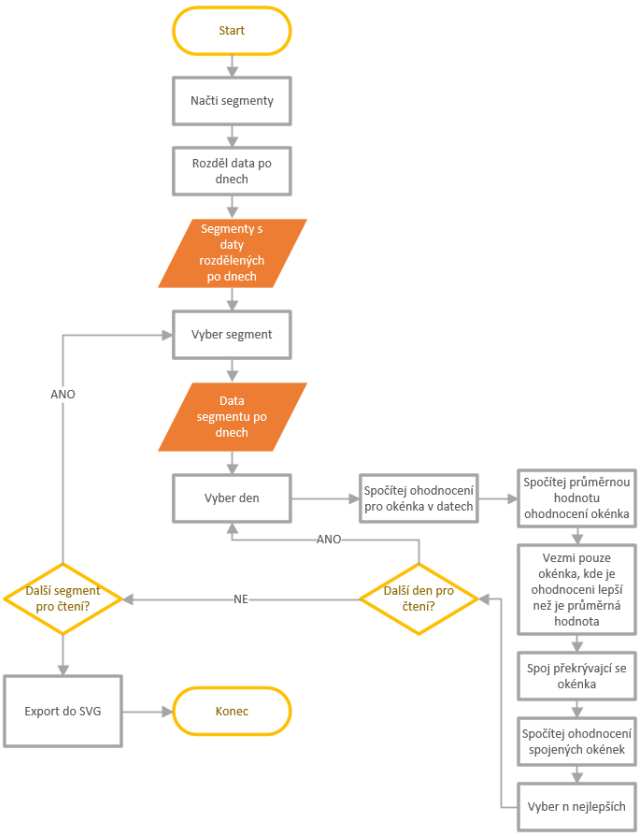
2.3 Paralelizace

Na obrázku 1 je možné vidět průběh výpočtu a interakcí programu. Slovně byl popsán v sekci 2.1.1

Pro parelelizaci této úlohy jsem si vybral paralelizaci se sdílenou pamětí a paralelizaci na GPGPU.

Pravděpodobně nejlepším řešením bude paralelizace na úrovni segmentů, tedy detekce kolísání jednotlivých segmentů bude probíhat současně. Důvodem výběru tohoto místa je, že se jedná o práci s větším množstvím nezávislých dat, tedy by režie spojená s paralelizací nemusel být zpomalující. Další možností paralelizace je paralelizace uvnitř segmentu na úrovni jednotlivých dní. Jednalo by se tak o paralelizaci uvnitř paralelizace (segment a jednotlivé dny). Vzhledem k tomu, že počet dat uvnitř jednotlivých dní je nízký (maximálně několik stovek), je předpokladem, že tato paralelizace přínos nepřinese. Tento přístup ale bude v rámci experimentu implementován.

Vzhledem k charakteru zvoleného algoritmu (mnoho podmínek, porovnávání) a obecně nízkému počtu dat, se kterými se provádí matematické operace, je předpokladem, že paralelizace na GPGPU bude spíše zdržující.



Obrázek 1: Flowdiagram programu

3 Programátorská dokumentace

V této kapitole bude popsána implementace programu řešícího zadanou úlohu. Popsány budou především důležité části programu. Program byl naprogramován v jazyce `C/C++` a byl vyvýjen v prostřední MS Visual Studio 2017. V rámci práce s pamětí byly využity smart pointery, získávaná data jsou reprezentována objekty. Při implementaci jsem využil verzovací systém Git.

3.1 Struktura zdrojových souborů

Zdrojové soubory jsou umístěny v adresáři LISKA_PPR. Zdrojové soubory jsou pak rozdělené logicky ještě do dalších složek podle funkcionality (Načítání vstupu, dat, hledání kolísání, export). Ve složce sqlLite jsou pak umístěny hlavičkové soubory knihovny SQLite. V této složce jsou také knihovny, které jsou potřebné programu a po překladu jsou nakopírované do patřičných složek.

3.2 Použité knihovny

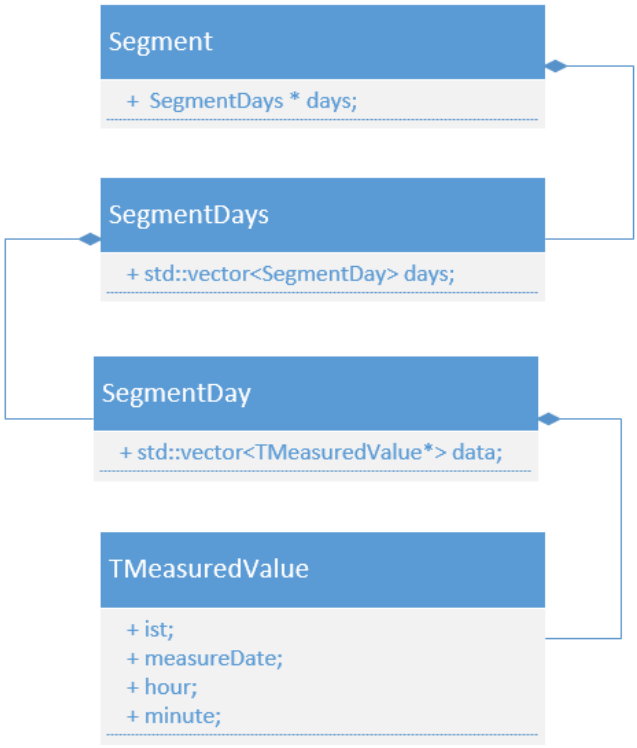
Pro práci se vstupními daty byla využita `SQLIte C/C++ Interface`, která umožňuje zadávat SQL dotazy nad jednotlivými tabulkami. Pro paralelizace pak bylo využito knihovny `TBB - Thread Building Blocks`, která je implicitně součástí `Intel Parallel Studio`. Pro paralelizaci na GPGPU bylo využito standardní knihovny AMP.

3.3 Spuštění aplikace

Zahájení aplikace probíhá v metodě `main()` v souboru `Main.cpp`. Po spuštění programu jsou vstupní parametry předány k parsování třídě `InputParser`. Pokud jsou vsechny parametry zadány korektně, je zahájen výpočet podle přepínače ve funkci `runSolution()`

3.4 Načtění dat

Načítání dat je implementováno v souboru `DataLoader.cpp`. Třída `DataLoader` obsahuje funkce a procedury, jak pro načtení dat z databáze, tak i jejich předzpracování, jako je nahrazení `NULL` hodnot. Pro načtení dat z databáze je zde využíváno funkcí knihovny `SQLIte C/C++ Interface`. Program očekává strukturu zmíněnou v zadání úlohy. Jedna naměřená jednotka je uložena ve třídě `TMEasuredValue`, kde se čas uložený v databázi přepočítává na čas, kterému rozumí běžný člověk. Naměřené jednotky konkrétního segmentu a dne pak tvoří data pro objekt `SegmentDay`. Dny pro konkrétní `Segment` jsou pak uloženy v objektu `SegmentDays`. Tímto způsobem jsou tedy data hierarchicky poskládána v objektu `Segment`. Struktura je popsána na obrázku 2



Obrázek 2: Hierarchie načtených dat

3.5 Hledání výkyvů

Algoritmus je vizualizován na obrázku 1.

Program byl nejprve naimplementován sériově a pak zparalelizován s využitím TBB. Jako poslední přišla na řadu paralelizace s využitím AMP. Vzhledem k tomu, že ve funkcích či metodách označených jako `restrict (amp)` jsou určitá omezení, jako např. absence pointerů, tříd a šablon, bylo nutné celou implementaci provést dvakrát.

Implementace pro procesor je tedy v souboru `PeakDetector.cpp` a amp implementace je v souboru `PeakDetectorAMP.cpp`. Parametry funkci zahajuji vypocet jsou ale totozna. Jedna se o velikost okénka, vstupní data a ukazatel na vektor objektů `Peak`, do kterého se vrací nalezená kolísání, rozdělená po dnech segmentů. Pokud je tedy segment tvořen 3 mi dny, je v kolekci odchylek na stejném indexu, jako je uložen segment, uložena kolekce o velikosti 3, kde je každá kolekce tvořena nalezenými kolísáními.

Implementace detekce v `PeakDetector.cpp` a `PeakDetectorAMP.cpp` je pak co se týče výsledků totožná. Liší se použitými strukturami jednotlivých funkcí.

3.5.1 Paralelizace na CPU

Pro paralelizace na úrovni jader procesoru byla použita knihovna `TBB - Thread Building Block`. Níže můžeme vidět konstrukci `tbb::parallel_for()`, která kód vepsaný uvnitř zparalelizuje. Je nutné zajistit zabránění konkurenčnímu zápisu ve sdílené proměnné, typicky že vlákno přistupuje k indexu v poli jako jedinné.

Implementace paralelizace v rámci CPU je realizovaná v rámci segmentů a navíc v rámci jednotlivých dní (pouze experimentálně). Zde je ukázka paralelizace s využitím TBB.

```
tbb::parallel_for(1,128,1,[=](int i){
   //Paralelni vypocet
}
```

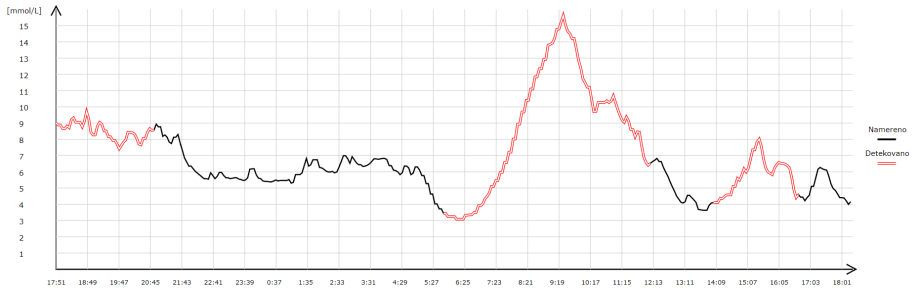
3.5.2 Paralelizace na GPGPU

Pro paralelizaci na GPGPU jsem použil knihovnu `C++ AMP`. Implementace je je totožná jako u CPU, nicméně před samotným výpočtem je nutné data zkopírovat do paměti grafiky, k čemuž se používá třída `concurrency::array_view<>`, která namapuje data do paměti GPGPU. Po dokončení výpočtu je potřeba zavolat metodu `synchronize()`, která vypropaguje změnu dat do pole v paměti. Zde je ukázka paralelizace na GPGPU.

```
//pole s hodnotami
std::vector<int> data;

... //naplneni pole data daty

//namapovani do pameti gpu
concurrency::array_view<int> data_view(data.size(),
    data.get());

//Amp parallel foreach
concurrency::parallel_for_each(data_view.extent,
    [=](concurrency::index<1> idx) restrict(amp)
{
   //vypocet paralelne
}
//vypropagovani dat mimo GPGPU
data_view.synchronize();
```

3.6 Vykreslení grafu

Vykreslení grafu probíhá, až když jsou všechna kolísání spočítána. Pro vykresleni grafu slouží třída `SVGExporter`, která podle přepínače vykreslí data do jednoho grafu, nebo data podle dní vykreslí do více grafů tak, že graf začíná v čase 0:00 a končí v čase 23:59. Výstup pak uloží do souboru [segment_číslo segmentu.svg]. Jednotlivá kolísání jsou pak do stejných grafů zaznamenána taktéž.



Obrázek 3: Ukázka vykresleného grafu

4 Získané výsledky

Testování a výpočty jsem prováděl na stolním počítači s procesorem `Intel Core i5-6500 @ 3,20GHz`, operační pamětí `RAM 16,0 GB`, s grafickou kartou `AMD RADEON R9 380 series 4 GB` a na operečním systému `Windows 10 64-bit`.

5 Doby běhů

Měření jsem provedl pro každou možnost (sériově, paralelně na úrovni segmentu, paralelně na úrovni segmentu a dne, GPGPU na úrovni segmentů) 1000 krát a časy jsem zprůměroval. Výsledné zprůměrované časy můžeme vidět v tabulce 1, časy v tabulce jsou uvedeny v ms (milisekundách).

Spolu s celkovou dobou jsem měřil i doby běhu paralelizované části na CPU, které jsou potřebné pro výpočet statistik (Amdahlův zákon). Program byl spuštěn s velikostí okénka 24, bez exportu grafů, a s výpisem do souboru.

Sériově	**Paralelně segmenty**	**Paralelně segmenty a dny**	**GPGPU**
131,21 ms	127,59 ms	128,071 ms	202,15 ms

Tabulka 1: Udává průměrné doby běhů, časy jsou uvedeny v ms (milisekundách)

Na datech můžeme pozorovat, že výpočet na akcelerátoru je zdaleka nejpomalejší. Nejlepšího výsledku dosahuje paralalelizace na úrovni segmentů. Paralelizace na úrovni segmentů a dnů nedopadla o moc hůře.

6 Reálné urychlení

Reálně urychlení se spočítá podle vztahu 1, kde T_1 je doba běhu sekvenční

Z naměřených hodnot můžeme určit reálné urychlení S pro jednotlivé implementace, které je dáno vztahem 1, kde T_1 je doba běhu sekvenční verze programu a T_p je doba běhu paralelizovaného kódu

$$S = \frac{T_1}{T_p} \tag{1}$$

Pokud je hodnota větší než jedna došlo k urychlení výpočtu pokud menší pak došlo ke zpomalení.

Paralelně segmenty	**Paralelně segmenty a dny**	**GPGPU**
1,03	1,02	0,65

Tabulka 2: Udává reálné urychlení jednotlivých verzí paralelizace oproti sekvenčnímu výpočtu

Z tabulky urychlení 2 můžeme vidět, že urychlení bylo dosaženo v případech paralelizace na úrovni segmentů a na úrovni segmentů a dnů na CPU. Při výpočtu na GPGPU došlo k výraznému zpomalení. Tímto se potvrdila hypotéza, že výpočet na GPGPU není pro tuto úlohu vhodný (málo dat, velká režie, nevhodný algoritmus pro GPU) U dalších výpočtů budu uvažovat, že P je počet procesorů roven 4.

7 Amdahlův zákon

Dobu běhu na jednom procesoru (sekvenčního výpočtu) T_1 si rozdělíme na dvě části podle 2, kde t_s označuje čas výpočtu na části kódu, kterou nelze paralelizovat a t_p čas výpočtu na kódu, který je paralelizovatelný, ale vykonaný sekvenčně.

$$T_1 = t_s + t_p \tag{2}$$

V ideálním případě pak bude při výpočtu na P procesorech čas výpočtu dán vztahem 3.

$$T_p = t_s + \frac{t_p}{P} \tag{3}$$

Po dosazení vyjádřených T_1 a T_p do vztahu 1 dostaneme Amdahlův zákon 4. Pokud si pomocí f označíme poměr času stráveného výpočtem sekvenčí části ku celkovému času dostaneme vzorec 5.

$$S = \frac{T_1}{T_p} = \frac{t_s + t_p}{t_s + \frac{t_p}{P}} \tag{4}$$

$$f = \frac{t_s}{t_s + t_p} \tag{5}$$

Dosazením a vyjádřením f do 1 získáme známější tvar Amdahlova zákona.

$$S = \frac{1}{f + \frac{1-f}{P}} \leq \frac{1}{f} \tag{6}$$

Pro výpočet f a S potřebujeme jednotlivé doby t_p a t_s , které jsou uvedeny v tabulce 3. Dobu t_s získáme odečtením celkové doby běhu z tabulky 1 od odpovídajících dob t_p.

Doba	**Sériově**
T_1	131,213 ms
t_p	2,81 ms
t_s	128,403 ms

Tabulka 3: Doby běhu programu sekvenčně

Dosazením získaných hodnot do vzorců 5 a 6 (popř. 4) získáme požadované statistické údaje f a S, které můžeme vidět v tab. 4.

	Sériově
f	$0,979$
S	$1,016$

Tabulka 4: Udává vypočítané hodnoty ukazatelů f a S pro Amdahlův zákon

8 Gustafsonův zákon

Dobu běhu na P procesorech označíme jako T_p danou vztahem 7, kterou rozdělíme na dobu běhu paralelní části t_p^* a sekvenční části t_s^*

$$T_p = t_s^* + t_p^* \tag{7}$$

Doba běhu na jednom procesoru bude dána vzorcem 8.

$$T_1 = t_s^* + P \cdot t_p^* \tag{8}$$

Dosazením do vzorce 1 pro zrychlení dostanem 9

$$S = \frac{T_1}{T_p} = \frac{t_s^* + P \cdot t_p^*}{t_s^* + t_p^*} \tag{9}$$

Opět můžeme vyjádřit α jako poměr času stráveného výpočtem sekvenčnčí části ku celkovému času, ale na paralelním počítači a dostaneme vzorec 10.

$$\alpha = \frac{t_s^*}{t_s^* + t_p^*} \tag{10}$$

Zrychlení pak také můžeme zapsat jako 11

$$S = P - \alpha(P - 1) \tag{11}$$

Jednotlivé doby t_s^* a t_p^* jsou vidět v tabulce 5

Doba	**TBB segment**	**TBB segment a den**	**Akcelerator**
T_p	127,59 ms	128,071 ms	202 ms
t_p^*	1,16 ms	1,26 ms	61,721 ms
t_s^*	126,43 ms	126,81 ms	140,279 ms

Tabulka 5: Udává jednotlivé doby běhu paraleně vykonaného programu

Dosazením do vzorce 10 a 11 získáme požadované ukazatele α a S, které jsou vypočteny v tabulce 6

	TBB segment	**TBB segment a den**	**Akcelerator**
α	$0,990$	$0,989$	$0,0766$
S	$1,027$	$1,033$	$1,702$

Tabulka 6: Udává vypočítané hodnoty ukazatelů α a S pro Gustafsonův zákon

9 Karp-Flattova metrika

Karp-Flatovu metriku e je definovaná vzorcem 12, kde P je počet procesorů a ψ je urychlení na P procesorech

$$e = \frac{\frac{1}{\psi} - \frac{1}{P}}{1 - \frac{1}{P}} \tag{12}$$

Pro výpočet ψ podle 13 potřebujeme znát $T(p)$, který určíme ze vztahu 14, kde T_s je doba běhu sekvenční části kódu a T_p je doba běhu paralelizovatelné části sekvenčně. Dále je potřeba určit $T(1)$ podle 15.

$$\psi = \frac{T(1)}{T(p)} \tag{13}$$

$$T(p) = T_s + \frac{T_p}{P} \tag{14}$$

$$T(1) = T_s + T_p \tag{15}$$

Vypočítané hodnoty lze vidět v tabulce 7, při výpočtu byly použity naměřené hodnoty z 3

	Sériově
ψ	$1,0269$
e	$0,723$

Tabulka 7: Udává vypočítané hodnoty ukazatelů ψ a e pro Karp-Flattovu metriku

10 Uživatelská příručka

Pro spuštění aplikace je potřeba mít v PC, na které aplikace poběží, dynamické knihovny pro `TBB - Thread Building Blocks` a `SQLLite`. SqlLite je přibalené v rámci odevzdané práce.

10.1 Překlad aplikace

Aplikace se překládá za využití vývojového prostředí `MS Visual Studio 2017`, ve kterém je možné přiložený projekt otevřít. Při překladu bude vývojové prostředí vyžadovat nastavení cest ke knihovnám `TBB` a `SQLite`.

10.2 Spuštění aplikace

Přeložený spustitelný soubor se jmenuje `LISKA_PPR.exe`. Aplikaci je možné spustit s využitím vývojového prostředí či z příkazové řádky příkazem `LISKA_PPR.exe [PARAMETRY]`. Jednotlivé parametry programu budou popsány v následující podsekci.

Program vypisuje standartní výstup. Data jsou ve formátu CSV a to ve formátu:
`Id segmentu;index dne;hodina zacatku kolisani:minuta zacatku kolisani; hodina konce kolisani:minuta konce kolisani`

10.2.1 Parametry programu

Program vyžaduje dva povinné parametry. Prvním je cesta a název souboru s daty. Druhým parametrem je metoda paralelizace, která má být použita.

```
Parametry:
-db [cesta k databazi]
-method [identifikator par. metody: 1 - seriove, 2 - paralelne
    na urovni semgnetu, 3 - paralelne na zaklade dnu, 4 -
    akcelerator]
-exportPath [kam se ma exportovat svg]
-window [velikost detekcniho okna]
-graphPerDay [identifikator zvoleneho grafu: 0 - Segment v
    jednom grafu, 1 - Co den, to graf]

Ukazka:
LISKA_PPR.exe -db "direcnet.sqlite" -method 2 -window 24
    -graphPerDay 0 -exportPath "/Export"
```

11 Závěr

Zadání semestrální práce bylo splněno.

Vytvořil jsem program pro detekci kolísání glukózy v naměřených datech. Při vypracování úlohy jsem se naučil pracovat se smart pointery, které pro mě byly novinkou a do budoucna je budu určitě využívat. Dále jsem se naučil, jak implementovat paralelizaci s využitím knihovny `TBB` a jak provádět výpočet na akceleráru s využitím `C++ AMP`.

Zvolená metoda pro detekci se na zkoumaných datech jeví jako dostačující. Jejími nedostatky je ovšem to, že nedokáže poznat, zda se data ve zkoumaném okně pouze vlní, nebo se významně mění.Tento problém by šel odstranit vyhlazením dat. Na to mi už ale nezbyl čas.

Program by šel rozšířit o další metody detekce kolísání. Zajímavými se mi zdály genetické algoritmy, či nějakým způsobem pracovat s integrály.