



Semestrální práce - Detekce pravděpodobnostního
rozdělení dat

Paralelní programování

Ondřej Drtina
A20N0077P
drtinao@students.zcu.cz

Obsah

1	Zadání	3
2	Analýza	4
2.1	Kolmogorov-Smirnov test	4
2.2	Shapiro-Wilk test	4
2.3	Chí-kvadrát test dobré shody	4
3	Popis implementace	5
3.1	Pravděpodobnostní rozdělení	5
3.1.1	NormalDistrib.h	5
3.1.2	UniformDistrib.h	6
3.1.3	ExponentialDistrib.h	6
3.1.4	PoissonDistrib.h	6
3.2	Chí-kvadrát test dobré shody	7
3.2.1	ChiSquareManager.h	7
3.2.2	IntervalManager.h	7
3.3	Správa výpočetních zařízení (OpenCL / SMP)	8
3.3.1	OpenCLManager.h	8
3.3.2	Farmer.h	8
3.3.3	cl_src.h	9
3.3.4	cl_defines.h	9
3.4	Spuštění a běh programu	9
3.4.1	Main.h	9
3.4.2	Initializer.h	10
3.4.3	Watchdog.h	10
3.5	Ostatní	10
3.5.1	DecisionDist.h	10
3.5.2	FileHelper.h	10
3.5.3	const.h	10
3.5.4	Structures.h	11
4	Vektorizované části programu	12
4.0.1	ChiSquareManager.cpp(řádka 130)	12
4.0.2	ChiSquareManager.cpp(řádka 148)	13
4.0.3	ChiSquareManager.cpp(řádka 493)	15
4.0.4	ChiSquareManager.cpp(řádka 502)	16
4.0.5	IntervalManager.cpp(řádka 41)	16

5	Části programu optimalizované pro OpenCL + SMP	18
5.0.1	První průchod algoritmu	18
5.0.2	Druhý průchod algoritmu	19
6	CFD + DFD	20
7	Spuštění programu	23
7.1	Požadavky	23
7.2	Spuštění	23
7.3	Interpretace výsledků	23
8	Benchmark	24
9	Závěr	25

1 Zadání

V rámci semestrální práce z předmětu PPR jsem si zvolil standardní zadání. Mým cílem tedy bylo implementovat algoritmus, který umožňuje zjistit, jakému pravděpodobnostnímu rozdělení jsou vstupní data nejbližší. Na implementaci jsou kladeny zejména následující požadavky:

- aplikace musí být paralelizována (SMP / OpenCL)
- využití paměti nesmí překročit 1GB
- součástí je watchdog vlákno, které hlídá správnou funkci programu
- program nemůže vytvářet nové soubory na disku
- program musí skončit do 15 minut

Pro úplnost dodávám, že po domluvě s vyučujícím bylo možno si v rámci daného předmětu zvolit i jiná zadání semestrální práce.

2 Analýza

Samotnému programování řešení úlohy předcházelo hledání metod, které se zaměřují na detekci pravděpodobnostního rozdělení vzorků dat. Rozhodl jsem se blíže prozkoumat následující metody: Kolmogorov-Smirnov test, Shapiro-Wilk test, chí-kvadrát test dobré shody a vyhodnotit jejich vhodnost pro řešení dané úlohy.

2.1 Kolmogorov-Smirnov test

Existuje ve dvou variantách - „one-sample test“ a „two-sample test“. Nejprve jsem myslel, že slouží pouze pro zjištění, zda existuje spojitost mezi pravděpodobnostním rozdělením dvou datasetů - odpovídá „two-sample test“. Je tedy zřejmé, že „two-sample test“ je pro danou úlohu nevhodný, jelikož nemáme k dispozici žádné vzorové rozdělení, se kterým bychom chtěli vstupní data srovnat.

One-sample test by na danou úlohu aplikovat šel, ale je určena primárně pro dataset s malým počtem čísel. Vzhledem ke zpracovávanému počtu čísel (soubor velký několik GB) mi metoda nepřišla vhodná a rozhodl jsem se implementovat jinou.

2.2 Shapiro-Wilk test

Metoda umožňuje zjistit pouze zda data pochází z normálního rozdělení. Využít by se pravděpodobně na zadanou úlohu dala, ale bylo by nutné implementovat i další metody pro zjištění, jakému rozdělení je vstupní dataset nejbližší. Hledal jsem univerzální řešení, které bude vhodné k paralelizaci.

2.3 Chí-kvadrát test dobré shody

Chí-kvadrát test je možno aplikovat na všechna zadaná rozdělení (normální, exponenciální, Poisson, rovnoměrné) a jeho kroky jsou shodné pro všechna rozdělení. Jednotný test je vyjma možnosti snadnější paralelizace výhodný i z toho důvodu, že je možné výsledky testu mezi sebou přímo porovnat, není nutné zavádění další metriky (p-hodnota, atd.).

3 Popis implementace

Níže jsou popsány jednotlivé soubory `.h` (headery), které obsahují definici funkcí použitých v aplikaci. Popis zde uvedený je pouze stručný, jelikož každá třída i funkce je v kódu vždy důkladně zdokumentována. V dodaném archivu se soubory `.h` a `.cpp` nachází ve složce `src`.

3.1 Pravděpodobnostní rozdělení

Následující text popisuje účel headerů specifikujících jednotlivá pravděpodobnostní rozdělení. Pro každé z požadovaných rozdělení je definován jiný header, program je tak v případě potřeby snadno rozšiřitelný pro detekci dalších rozdělení.

3.1.1 NormalDistrib.h

Obsahuje definici funkcí, jež se vztahují ke Gaussovu (normálnímu) rozdělení. Konstruktor očekává pouze dvě hodnoty:

- průměrnou hodnotu datasetu,
- směrodatnou odchylku datasetu.

Hodnoty předané v konstruktoru jsou následně využity pro výpočet distribuční funkce.

Výpočet distribuční funkce normálního rozdělení vyžaduje řešení integrálů, s čímž jsem se vypořádal tak, že nejprve provedu normalizaci vstupních dat a následně hledám hodnotu distribuční funkce v poli `standardize_dist_arr`.

Pole obsahuje 4501 prvků, dochází tedy k zaokrouhlování na nejbližší odpovídající hodnotu. Pro účely této aplikace se zdá být počet prvků dostatečný. Testovací data (dodána ve složce `referencni_rozdeleni`) jsou rozpoznána korektně. Rovněž jsem zkoušel vygenerovat větší soubory pomocí generátoru, program fungoval dle očekávání. Samozřejmě by bylo možné pole rozšířit a dosáhnout tak o něco přesnějších výpočtů, ovšem za cenu využití většího množství operační paměti.

Na výpočet distribuční funkce normálního rozdělení nejsou kladena žádná omezení. Vstupní data mohou obsahovat jakoukoliv kombinaci čísel (desetinná čísla / záporná čísla). Výpočet chí-kvadrát testu dobré shody je tedy proveden při každém validním spuštění programu.

3.1.2 UniformDistrib.h

Uvedený header obsahuje definici funkcí umožňující výpočet distribuční funkce rovnoměrného rozdělení. Konstruktor, i v tomto případě, očekává právě dvě hodnoty:

- minimální hodnotu nacházející se v datasetu,
- maximální hodnotu nacházející se v datasetu.

Vzorec pro výpočet distrib. funkce rovnoměrného rozdělení je přímočarý a pracuje pouze s hodnotami předanými v konstruktoru a hodnotou intervalu.

Vstupní data nemusí, pro testování náležitosti do rovnoměrného rozdělení, splňovat žádná kritéria. Stejně jako v případě normálního rozdělení se tedy může jednat o libovolnou kombinaci čísel.

3.1.3 ExponentialDistrib.h

Umožňuje výpočet distribuční funkce exponenciálního rozdělení, konstruktor, v tomto případě, očekává pouze průměrnou hodnotu datasetu.

Na rozdíl od předchozích uvedených rozdělení, exponenciální rozdělení klade omezení na hodnoty vyskytující se ve vstupním datasetu. Ke zjištění, zda hodnoty v datasetu odpovídají exponenciálnímu rozdělení, musí být všechny hodnoty pouze kladné. V datasetu se mohou vyskytovat i desetinná čísla.

3.1.4 PoissonDistrib.h

Hlavičkový soubor obsahuje definici funkce pro výpočet distribuční funkce Poissonova rozdělení. Konstruktor očekává, stejně jako v případě exponenciálního rozdělení, průměrnou hodnotu datasetu.

Kromě funkce, která vypočítá hodnotu distribuční funkce pro jedno konkrétní číslo, se zde nachází i funkce, jež vypočítá funkci pro všechny hodnoty v intervalu. Jedná se o funkci `calc_distrib_func_interval(long lower_boundary, long upper_boundary)`, jež očekává spodní a horní hranici intervalu. Výpočet je následně proveden pro všechna odpovídající celá čísla.

Z předchozího odstavce je zřejmé, že na výpočet distribuční funkce Poissonova rozdělení jsou kladeny striktní požadavky. Vstupní dataset musí obsahovat pouze celá, kladná čísla. V opačném případě dataset nereprezentuje Poissonovo rozdělení.

3.2 Chí-kvadrát test dobré shody

Následující text popisuje hlavičkové soubory související s chí-kvadrát testem, jež je použit pro zjištění, k jakému rozdělení mají vstupní data nejblíže.

3.2.1 ChiSquareManager.h

Obsahuje funkce specifické pro výpočet chí-kvadrát testu dobré shody, přítomné funkce umožňují zejména:

- výpočet očekávané pravděpodobnosti
- výpočet očekávané četnosti (frekvence)
- vyhodnocení vzorce specifického pro chí-kvadrát $(n_i - \text{expectFreq})^2 / \text{expectFreq}$
 - `ni` = počet čísel vyskytujících se v daném intervalu
 - `expectFreq` = očekávaná četnost pro daný interval
- výpočet testového kritéria
 - pro každou z uvažovaných distribucí je testové kritérium rovno součtu výsledků chí-kvadrát vzorce (napříč intervaly)

Pro většinu ze zmíněných funkcí je přítomný i ekvivalent, jež provede daný výpočet pro všechny definované intervaly. Např. funkce `std::vector<double> calc_expected_prob_bulk(std::vector<double> d_func_res)` provede výpočet očekávané pravděpodobnosti pro všechny intervaly nacházející se v předaném vektoru.

Jelikož intervalů je zpravidla více a prováděné operace jsou pro každý interval shodné, je část kódu v této třídě vhodná k vektorizaci. Konkrétně se mi podařilo vektorizovat 4 smyčky.

V tomto headeru jsou i hlavičky funkcí, které umožňují vypsát obsah struktur obsahujících data testu v uživatelsky přívětivé formě. Například funkce `void print_chi_crit_res(chi_crit_res_struct* chi_crit_res)` zobrazí výsledek testového kritéria pro každou z distribucí.

3.2.2 IntervalManager.h

Z předchozí kapitoly je vidno, že při vyhodnocení chí-kvadrát testu jsou vstupní data dělena do intervalů. Uvedené řešení je samozřejmě spojeno se ztrátou části původních dat. Pro výpočet chí-kvadrát testu však není uchování všech hodnot žádoucí a ani možné (program limitován na 1GB RAM).

Header tedy obsahuje definici funkcí, jež se starají o správu intervalů v programu (vytvoření intervalů o dané velikosti, atd.). Vhodný počet intervalů získávám výpočtem Sturgesova pravidla, jež je definováno jako $k = 1 + 3,32 \times \log(n)$, kde n je počet validních čísel v datasetu.

Vytvoření známého počtu intervalů o předem známé velikosti je samozřejmě také činnost, jež se dá vektorizovat.

Samotné dělení dat do intervalů je děleno mezi OpenCL / SMP zařízení a odpovídající funkce jsou v jiných souborech `Farmer.h`.

3.3 Správa výpočetních zařízení (OpenCL / SMP)

Hlavičkové soubory uvedené níže obsahují definice funkcí, které interagují se zařízeními umožňující paralelní výpočty (OpenCL / SMP).

3.3.1 OpenCLManager.h

Poskytuje definice funkcí, jež souvisí s OpenCL zařízeními. Umožňuje tedy např.:

- detekci všech OpenCL platforem + zařízení v systému
- inicializaci zařízení pro výpočet
 - vytvoření kontextu zařízení
 - kompilaci programu ze zdrojového kódu
 - vytvoření kernelu zařízení
 - vytvoření fronty zařízení
 - alokaci potřebných bufferů
- kontrolu, zda zařízení s daným názvem v systému existuje
- výpis zařízení v systému

3.3.2 Farmer.h

Obsahuje funkce, jež umožňují přidělit práci zařízení v systému. Definovaná třída tedy spravuje jednotlivé Workery v systému (model Farmer - Worker). Workerem může být buď OpenCL zařízení nebo SMP nebo obojí zmíněné (výpočet ALL).

Pro výpočet na SMP / OpenCL zařízení jsem optimalizoval následující části chí-kvadrát testu dobré shody (jsou nutné dva průchody souborem).

- první průchod (SMP / OpenCL)
 - nalezení minimální hodnoty datasetu
 - nalezení maximální hodnoty datasetu
 - kontrola, zda obsahuje zápornou hodnotu
 - kontrola, zda obsahuje desetinné číslo (= vynechat Poisson)
- druhý průchod (SMP / OpenCL)
 - zařazení čísla do určitého intervalu

Přítomné funkce se pokusí najít nejméně vytížené zařízení v systému a tomu úlohu přidělit. Vybíráno je vždy jen z povolených zařízení (pokud není výpočet ALL).

3.3.3 `cl_src.h`

Obsahuje funkce napsané pro OpenCL zařízení.

3.3.4 `cl_defines.h`

Specifikuje cílovou verzi OpenCL zařízení (2), povoluje OpenCL extensions (atomické operace).

3.4 Spuštění a běh programu

Uvedené hlavičkové soubory definují funkce stěžejní pro správnou inicializaci a běh programu.

3.4.1 `Main.h`

Definuje vstupní bod programu. Při spuštění s validními hodnotami dojde k volání funkce zajišťující první průchod souborem - funkce `perf_first_pass` a vypsání získaných hodnot `print_first_pass_info`. Obdobný postup je aplikován u druhého průchodu souborem. Výpočet chí-kvadrát testu se provede voláním funkce `perform_chi_square_calc`.

3.4.2 Initializer.h

Obsahuje definice funkcí, které zjišťují, zda byl program správně inicializován.

3.4.3 Watchdog.h

Watchdog hlídá správnou funkci programu a běží ve vlastním vlákne. Využil jsem konceptu timer watchdog. Tedy watchdogu běží odpočet, který musí být včas resetován některým z běžících vláken. Pokud se tak nestane, je běh programu vyhodnocen jako nevyhovující a watchdog začne vypisovat chybovou hlášku. Uživatel následně může program restartovat či podstoupit riziko a použít výsledky programu i přes zjištění chyby během výpočtu.

Odpočet je resetován zpravidla při vykonání nějaké činnosti, která je od programu žádoucí, tzn. např. proběhl výpočet distribuční funkce. Povolena je pouze jedna instance watchdogu, timeout je možno směnit v souboru `const.h`.

3.5 Ostatní

Kapitola popisuje headery, které nespádají do žádné z předchozích popisovaných kategorií.

3.5.1 DecisionDist.h

Spravuje data, jež jsou využita k vyhodnocení, zda daný dataset může reprezentovat určité rozdělení. Např. pokud soubor obsahuje zápornou hodnotu, nereprezentuje dataset Poisson ani exponenciální rozdělení. Rovněž obsahuje průměrnou hodnotu datasetu, směrodatnou odchylku, atd.

3.5.2 FileHelper.h

Definice funkcí, které souvisejí se čteným souborem. Umožňuje čtení ze souboru, kontrolu, zda je soubor přístupný, apod.

3.5.3 const.h

Konstanty měnící chování programu. Definuje například max. počet čísel, jež jsou najednou čteny ze souboru a timeout watchdoga.

3.5.4 Structures.h

Obsahuje struktury a výčtové typy, jež jsou využity pro předávání informací napříč programem. Ze struktur např.:

- `cl_dev_stuff_struct`
 - uchovává informace o OpenCL zařízení
 - kernel, fronta, kontext, kompilované programy, atd.
- `chi_part_res_struct`
 - uchovává částečné výsledky chí-kvadrát testu
 - vektor s výsledky distribuční funkce pro každé rozdělení

Výčtové typy, jež jsou definovány:

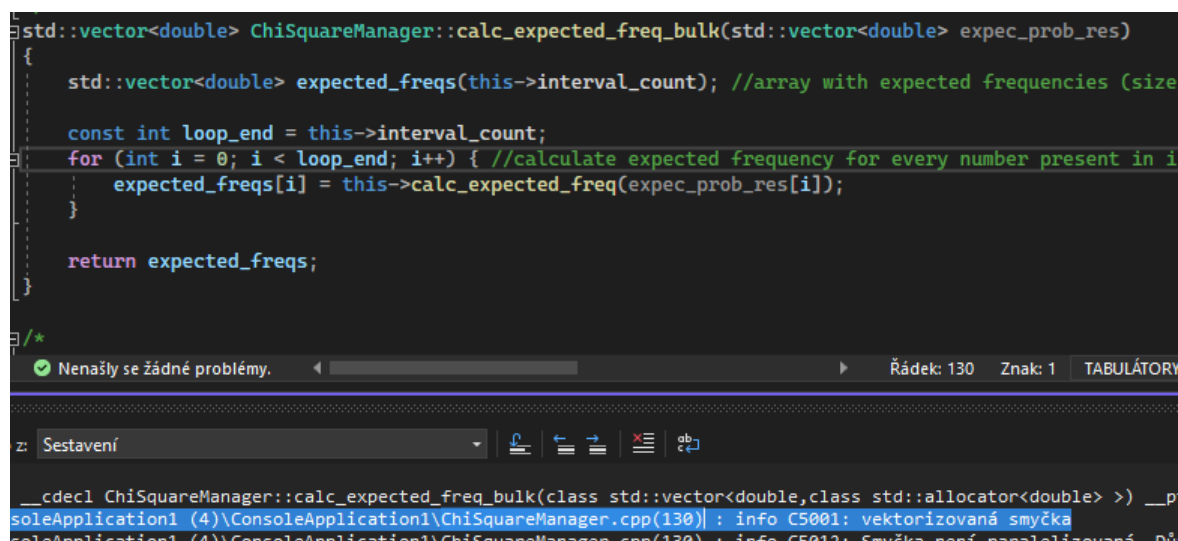
- `compute_type`
 - typ výpočtu, hodnoty: ALL, SMP, OPENCL
- `distribution_list`
 - seznam pravděpodobnostních rozdělení, jež je program schopný vyhodnotit
- `distribution_limit`
 - popisuje limitaci danou datasetem
 - např. NEGATIVE - v souboru je záporné číslo

4 Vektorizované části programu

Kapitola obsahuje popis částí programu, jež se mi podařilo vektorizovat pomocí autovektorizace (kompilátor Visual Studio). Celkově se mi podařilo zvektorizovat 5 for cyklů. U každého cyklu přikládám snímek obrazovky z VS, který informuje o autovektorizaci cyklu. U prvních dvou cyklů přikládám i část vygenerovaného assembly kódu, ze kterého je zřejmé rozbalení cyklu (zkratka CTRL + ALT + D během debugu ve VS).

4.0.1 ChiSquareManager.cpp(řádka 130)

Na řádce 130 se nachází for cyklus, jehož produktem je očekávaná frekvence pro každý z předaných intervalů. Jelikož očekávaná frekvence intervalů je na sobě nezávislá, podařilo se mi danou činnost autovektorizovat.



```
std::vector<double> ChiSquareManager::calc_expected_freq_bulk(std::vector<double> expec_prob_res)
{
    std::vector<double> expected_freqs(this->interval_count); //array with expected frequencies (size
    const int loop_end = this->interval_count;
    for (int i = 0; i < loop_end; i++) { //calculate expected frequency for every number present in i
        expected_freqs[i] = this->calc_expected_freq(expec_prob_res[i]);
    }
    return expected_freqs;
}
/*
Nenašly se žádné problémy.
Řádek: 130 Znak: 1 TABULÁTOR
Sestavení
__cdecl ChiSquareManager::calc_expected_freq_bulk(class std::vector<double,class std::allocator<double> >) __p
soleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(130) : info C5001: vektorizovaná smyčka
soleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(130) : info C5012: Smyčka není paralelizovaná. Dů
```

Obrázek 4.1: ChiSquareManager.cpp(řádka 130)

```

    for (int i = 0; i < loop_end; i++) { //calculate expected frequency for every number present
00007FF7393015E5 test     edx,edx
00007FF7393015E7 jle     ChiSquareManager::calc_expected_freq_bulk+15Fh (07FF7393016AFh)
00007FF7393015ED cmp     edx,8
00007FF7393015F0 jb     ChiSquareManager::calc_expected_freq_bulk+15Fh (07FF7393016AFh)
    expected_freqs[i] = this->calc_expected_freq(expec_prob_res[i]);
00007FF7393015F6 mov     r9,qword ptr [r14]
00007FF7393015F9 mov     r8,qword ptr [rsi]

    const int loop_end = this->interval_count;
00007FF7393015FC movd    xmm0,dword ptr [rdi]
00007FF739301600 pshufd  xmm0,xmm0,0
00007FF739301605 lea     eax,[rdx-1]
00007FF739301608 movsxd  rcx,eax
00007FF73930160B lea     r11,[r8+rcx*8]
00007FF73930160F lea     rax,[r9+rcx*8]
00007FF739301613 cmp     r8,rax
00007FF739301616 ja     ChiSquareManager::calc_expected_freq_bulk+0D1h (07FF739301621h)
00007FF739301618 cmp     r11,r9
00007FF73930161B jae     ChiSquareManager::calc_expected_freq_bulk+15Fh (07FF7393016AFh)
00007FF739301621 cmp     r8,rdi
00007FF739301624 ja     ChiSquareManager::calc_expected_freq_bulk+0DFh (07FF73930162Fh)
00007FF739301626 cmp     r11,rdi
00007FF739301629 jae     ChiSquareManager::calc_expected_freq_bulk+15Fh (07FF7393016AFh)
00007FF73930162F mov     ecx,edx
00007FF739301631 and     ecx,80000007h
00007FF739301637 jge     ChiSquareManager::calc_expected_freq_bulk+0F0h (07FF739301640h)
00007FF739301639 dec     ecx
00007FF73930163B or     ecx,0FFFFFFF8h
00007FF73930163E inc     ecx
00007FF739301640 mov     eax,edx
00007FF739301642 sub     eax,ecx
    for (int i = 0; i < loop_end; i++) { //calculate expected frequency for every number present
00007FF739301644 movsxd  r11,eax
00007FF739301647 mov     rcx,rbp
    expected_freqs[i] = this->calc_expected_freq(expec_prob_res[i]);
00007FF73930164A movq    xmm0,xmm0
00007FF73930164E cvtdq2pd  xmm2,xmm0

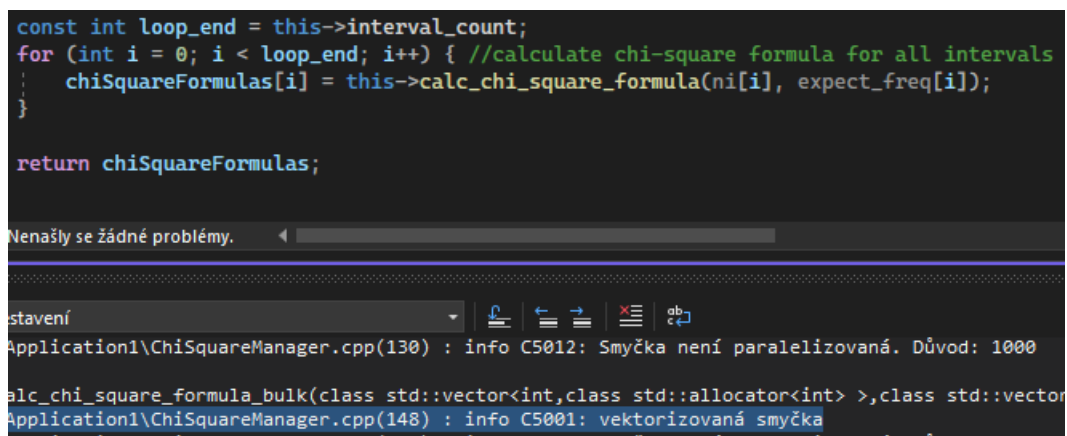
    const int loop_end = this->interval_count;
00007FF739301652 mov     eax,20h
00007FF739301657 nop     word ptr [rax+rax]
    expected_freqs[i] = this->calc_expected_freq(expec_prob_res[i]);
00007FF739301660 mov     rax,rax

```

Obrázek 4.2: ChiSquareManager.cpp(řádka 130)

4.0.2 ChiSquareManager.cpp(řádka 148)

Řádka 148 obsahuje `for` smyčku, která řeší výpočet formule chí-kvadrát testu pro předané intervaly. Opět se jedná o nezávislou činnost, provedení autovektorizace je tedy žádoucí.



Obrázek 4.3: ChiSquareManager.cpp(řádka 148)

```

Možnosti zobrazení
for (int i = 0; i < loop_end; i++) { //calculate chi-square formula for all intervals
00007FF676E31870 test     edx,edx
00007FF676E31872 jle      ChiSquareManager::calc_chi_square_formula_bulk+148h (07FF676E31918h)
00007FF676E31878 cmp      edx,4
00007FF676E3187B jb       ChiSquareManager::calc_chi_square_formula_bulk+148h (07FF676E31918h)
    chiSquareFormulas[i] = this->calc_chi_square_formula(ni[i], expect_freq[i]);
00007FF676E31881 mov     r10,qword ptr [rsi]
00007FF676E31884 mov     r11,qword ptr [r14]
00007FF676E31887 mov     r8,qword ptr [rdi]

    const int loop_end = this->interval_count;
00007FF676E3188A lea     eax,[rdx-1]
00007FF676E3188D movsxd  rcx,eax
00007FF676E31890 lea     rbx,[r8+rcx*8]
00007FF676E31894 lea     rax,[r11+rcx*4]
00007FF676E31898 cmp     r8,rax
00007FF676E3189B ja      ChiSquareManager::calc_chi_square_formula_bulk+0D2h (07FF676E318A2h)
00007FF676E3189D cmp     rbx,r11
00007FF676E318A0 jae     ChiSquareManager::calc_chi_square_formula_bulk+148h (07FF676E31918h)
00007FF676E318A2 lea     rax,[r10+rcx*8]
00007FF676E318A6 cmp     r8,rax
00007FF676E318A9 ja      ChiSquareManager::calc_chi_square_formula_bulk+0E0h (07FF676E318B0h)
00007FF676E318AB cmp     rbx,r10
00007FF676E318AE jae     ChiSquareManager::calc_chi_square_formula_bulk+148h (07FF676E31918h)
00007FF676E318B0 mov     ecx,edx
00007FF676E318B2 and     ecx,80000003h
00007FF676E318B8 jge     ChiSquareManager::calc_chi_square_formula_bulk+0F1h (07FF676E318C1h)
00007FF676E318BA dec     ecx
00007FF676E318BC or      ecx,0FFFFFFFh
00007FF676E318BF inc     ecx
00007FF676E318C1 mov     eax,edx
00007FF676E318C3 sub     eax,ecx
    for (int i = 0; i < loop_end; i++) { //calculate chi-square formula for all intervals
00007FF676E318C5 movsxd  rcx,eax
00007FF676E318C8 mov     rax,r15
00007FF676E318CB nop     dword ptr [rax+rax]
    chiSquareFormulas[i] = this->calc_chi_square_formula(ni[i], expect_freq[i]);
00007FF676E318D0 movups  xmm2,xmmword ptr [r10+rax*8]
00007FF676E318D5 cvtdq2pd xmm1,mmword ptr [r11+rax*4]
00007FF676E318DB sub     xmm1,xmm2
}

```

Obrázek 4.4: ChiSquareManager.cpp(řádka 148)

4.0.3 ChiSquareManager.cpp(řádka 493)

Vektorizace for cyklu, jež provádí výpočet distribuční funkce rovnoměrného rozdělení pro každý z předaných intervalů. Jedná se o nezávislou činnost - vhodná k vektorizaci.


```

for (int i = 0; i < loop_end; i++) { //calculate uniform distrib. func. for all created intervals
    uniform_res_arr[i] = uniformDistrib->calc_distrib_func(interval_bound_up[i] / decisionDist->get_max_val
}
distrib_func_res->uniform_res_arr = uniform_res_arr; //add uniform results to struct

```

nenášly se žádné problémy. Řádek: 493

stavení

ChiSquareManager::calc_distrib_func(class IntervalManager * __ptr64, class DecisionDist * __ptr64) __ptr64
 n1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(493) : info C5001: vektorizovaná smyčka

Obrázek 4.5: ChiSquareManager.cpp(řádka 493)

4.0.4 ChiSquareManager.cpp(řádka 502)

Zde je vektorizován for cyklus, který provádí převod každého z intervalů normální normované rozdělení. Opět se jedná o nezávislou činnost.

```

std::vector<double> standard_val_res(interval_count);
for (int i = 0; i < loop_end; i++) { //standardize value for each interval
    standard_val_res[i] = normalDistrib->standardize_interval(interval_bound_up[i] / decisionDist->get_max_value()); //no
}

```

nenášly se žádné problémy. Řádek: 493 Znak: 2 Sloupec: 5 TAB

stavení

+.\ConsoleApplication1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\Farmer.cpp(470) : info C5012: Smyčka není paralelizovaná. Důvod: 1007
 +.\Community\VC\Tools\MSVC\14.33.31629\include\xmemory(1791) : info C5002: Smyčka není vektorizovaná. Důvod: 1301
 +.\Community\VC\Tools\MSVC\14.33.31629\include\xmemory(1791) : info C5012: Smyčka není paralelizovaná. Důvod: 1007

* __ptr64 __cdecl ChiSquareManager::calc_distrib_func(class IntervalManager * __ptr64, class DecisionDist * __ptr64) __ptr64
 +.\ConsoleApplication1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(493) : info C5001: vektorizovaná smyčka
 +.\ConsoleApplication1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(502) : info C5001: vektorizovaná smyčka
 +.\ConsoleApplication1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\ChiSquareManager.cpp(502) : info C5001: vektorizovaná smyčka

Obrázek 4.6: ChiSquareManager.cpp(řádka 502)

4.0.5 IntervalManager.cpp(řádka 41)

Uvnitř tohoto cyklu se definují vektory, jejichž obsah reprezentuje hranice jednotlivých intervalů. Je zřejmé, že hranice intervalů jsou na sobě nezávislé.

```
const int loop_end = this->interval_count;
std::vector<double> interval_low_vect(this->interval_count,0);
std::vector<double> interval_up_vect(this->interval_count,0);
for (int i = 0; i < loop_end; i++) { //define boundaries for each interval, save it to vector
    interval_low_vect[i] = ((this->min_value_data / this->max_value_data) + (this->interval_size * i)) * this->max_value_data; //lower
    interval_up_vect[i] = ((this->min_value_data / this->max_value_data) + (this->interval_size * (i + 1))) * this->max_value_data; //upper
}
//calculated values - END
this->interval_bound_low = interval_low_vect;
this->interval_bound_up = interval_up_vect;

this->interval_size *= this->max_value_data; //boundaries calculated, get original number
```

nenášly se žádné problémy. Řádek: 17 Znak: 22

stavení

Application1 (4)\ConsoleApplication1 (3)\ConsoleApplication1 (4)\ConsoleApplication1\IntervalManager.cpp(41) : info C5001: vektorizovaná smyčka
C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.32.31520\include\memory(1701) : info C5012: Smyčka není vektorizovaná. Důvod: 1002

Obrázek 4.7: IntervalManager.cpp(řádka 41)

5 Části programu optimalizované pro OpenCL + SMP

Níže je popis částí programu, které jsou uzpůsobeny pro paralelní běh na OpenCL / SMP zařízeních.

5.0.1 První průchod algoritmu

Popisuje využití OpenCL / SMP zařízení při prvním průchodu souborem.

Využití workera

Při prvním průchodu algoritmu jsou OpenCL a SMP zařízení využita k nalezení minima + maxima datasetu a zjištění, zda dataset obsahuje desetinná či záporná čísla. Kód psaný pro OpenCL využívá atomické operace (více jader hledá min / max), jiné speciální konstrukce nejsou použity. SMP kód využívá knihovny Intel TBB a její konstrukce `parallel_for`.

SMP kód prvního průchodu je zahrnut ve funkci `smp_min_max_dec_point_neg_num` ve třídě `Farmer`. OpenCL kód je dostupný v headeru `cl_src.h`.

Práce s pamětí workera

V případě OpenCL jsou průběžná data (min, max atd.) uchovávána na zařízení a k jejich přečtení dojde až po skončení prvního průchodu. Potřebujeme znát až finální hodnotu. V případě TBB jsou data vrácena pro každou dokončenou úlohu - není mi znám způsob, který by umožnil TBB uchovávat hodnoty napříč tasky.

Sjednocení hodnot OpenCL + SMP

Funkce `retr_min_max_dec_point_neg_num_res` ve třídě `Farmer` přečte hodnoty z OpenCL zařízení a srovná je s hodnotami získanými SMP (TBB). Jako minimální hodnota datasetu je vyhodnocena nejmenší hodnota získaná napříč OpenCL / SMP, v případě maximální hodnoty je postup analogický. Pro vyhodnocení, zda soubor obsahuje záporné číslo stačí, aby odpovídající flag mělo nastavené libovolné zařízení.

5.0.2 Druhý průchod algoritmu

Níže je rozepsáno využití multiprocessorových zařízení při druhém průchodu algoritmu.

Využití workera

Při druhém průchodu souborem jsou zařízení využita pro řazení čísel do odpovídajících intervalů.

SMP kód lze vidět ve funkci `smp_add_nums_to_intervals` ve třídě `Farmer`. OpenCL kód je opět v headeru `cl_src.h`.

Práce s pamětí workera

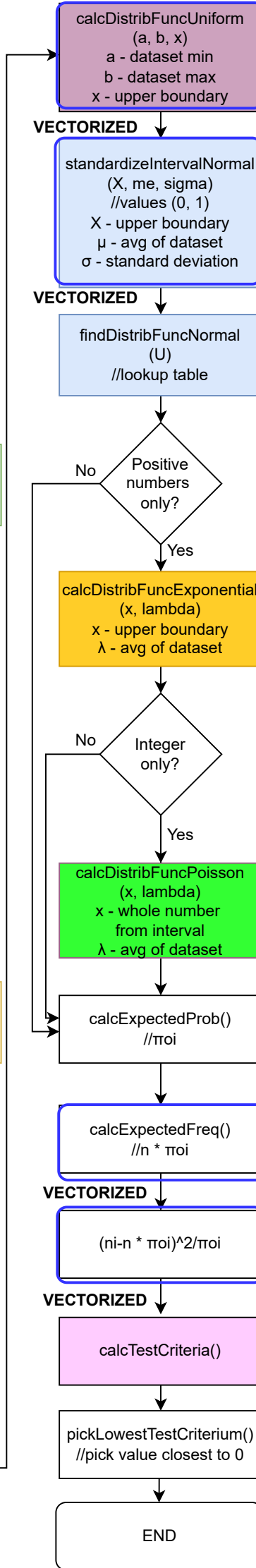
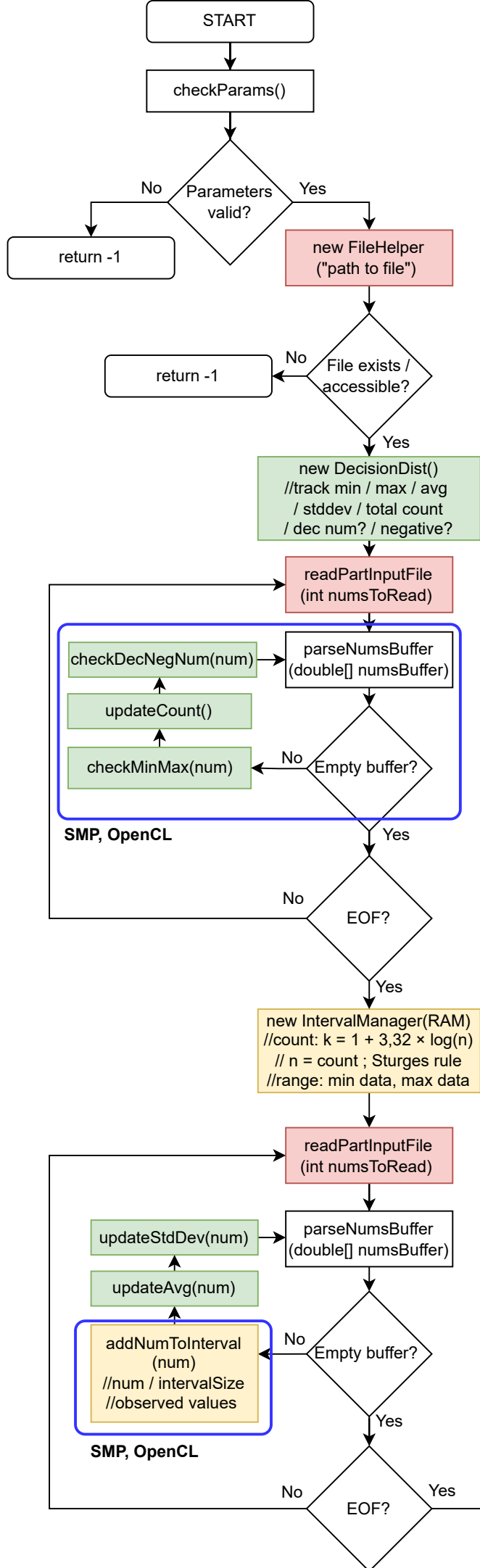
Situace je analogická jako u prvního průchodu. OpenCL zařízení si uchovávají počet čísel pro každý interval v paměti až do dokončení čtení souboru a následně jsou ze zařízení přečteny finální hodnoty pro každý interval. Ze SMP zařízení jsou hodnoty získávány ihned po dokončení přidělené úlohy.

Sjednocení hodnot OpenCL + SMP

Funkce `retr_add_nums_to_intervals_res` ve `Farmer` přečte hodnoty z OpenCL zařízení a sečte hodnoty pro každý interval získané napříč všemi OpenCL zařízeními + SMP.

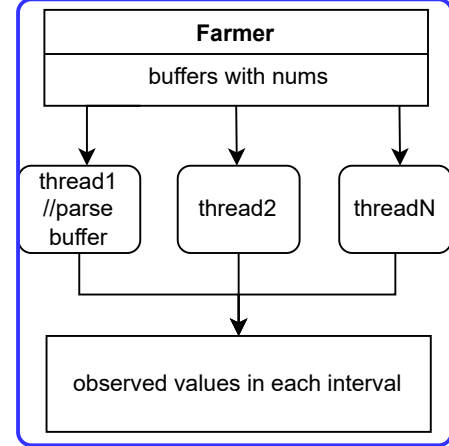
6 CFD + DFD

Následující stránky obsahují CFD a DFD diagramy. V CFD jsou znázorněny části, jež jsou paralelizovány.



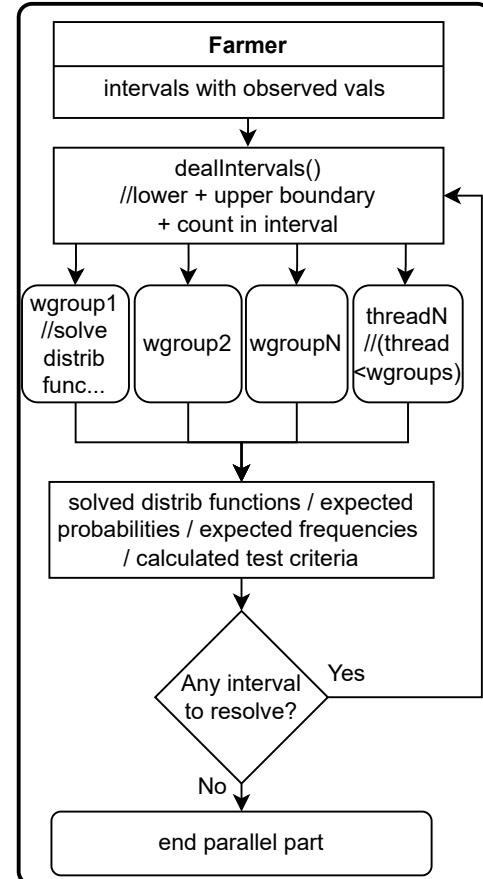
$$F(x) = \begin{cases} 0 & x \leq a \\ \frac{x-a}{b-a} & x \in (a, b) \\ 1 & x \geq b \end{cases}$$

$$U = \frac{X - \mu}{\sigma}$$

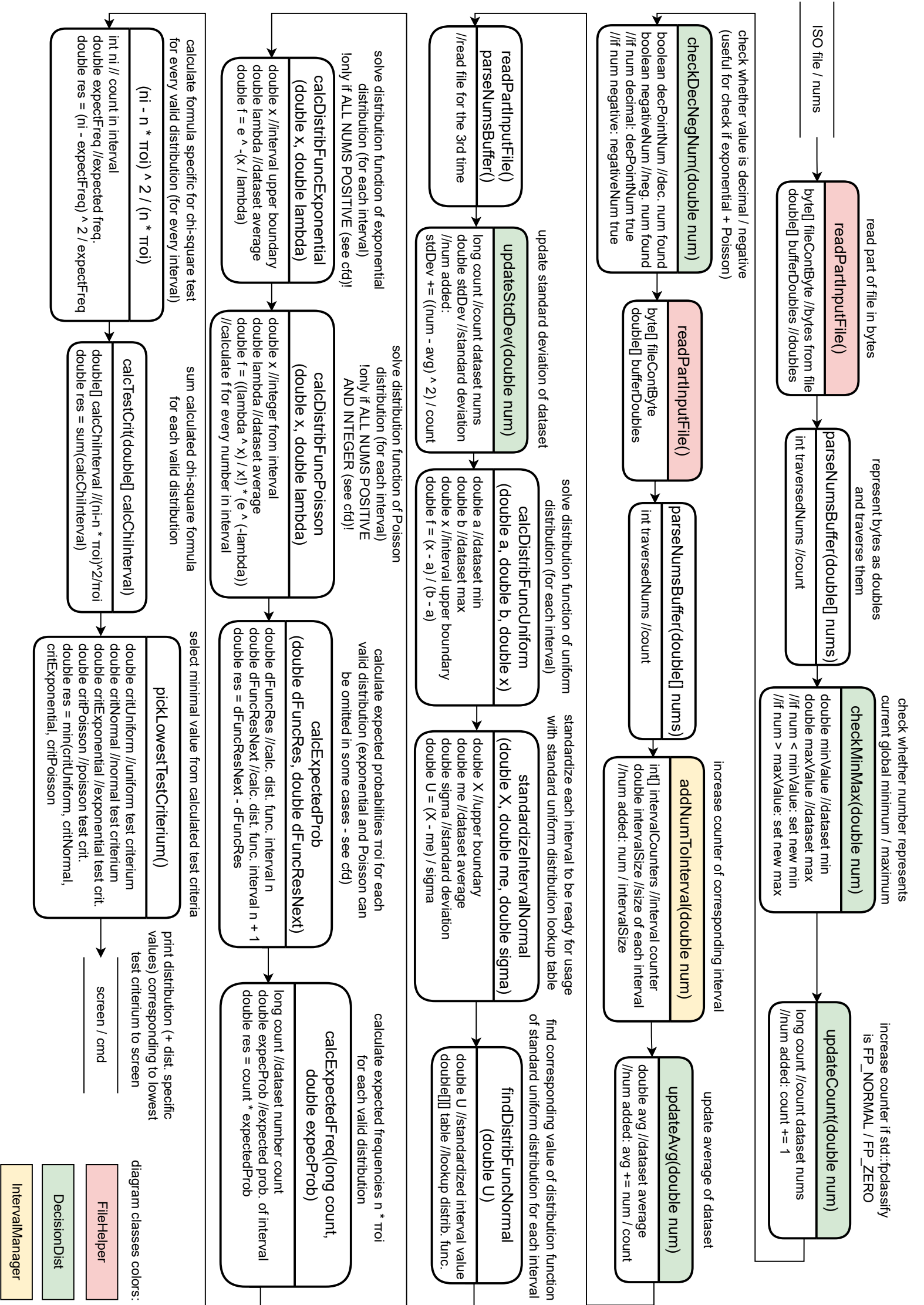


$$F(x) = e^{-\left(\frac{x}{\lambda}\right)}$$

$$F(x) = \sum_{x_i \leq x} \frac{\lambda^{x_i}}{x_i!} e^{-\lambda}$$



$$\sum_{i=1}^k \frac{(n_i - n\pi_{0,i})^2}{n\pi_{0,i}}$$



7 Spuštění programu

Obsahuje návod pro spuštění programu.

7.1 Požadavky

Pro využití OpenCL výpočtu je nutné vlastnit zařízení s OpenCL 2.0, které podporuje 64-bit atomické instrukce.

7.2 Spuštění

Dle požadavků se program spouští příkazem pprsolver.exe soubor processor.

7.3 Interpretace výsledků

Pro každé validní rozdělení je vypsána hodnota chí-kvadrát testového kritéria. Rozdělení s nejmenší hodnotou testového kritéria je danému datasetu nejbližší.

Program vypisuje celou workflow chí-kvadrát testu, tedy před vypsáním finálního rozdělení jsou ukázány intervaly, do kterých jsou data rozdělena, poté jsou ukázány výsledky distribučních funkcí atd.

```
****CALCULATED CHI-SQUARE TEST CRITERIA*** START
uniform result: 3343.94
normal result: 227.982
exponential result: 22.2938
Poisson result: 5.03259
****CALCULATED CHI-SQUARE TEST CRITERIA*** END
****CLOSEST DISTRIBUTION INFO*** START
closest distribution is: Poisson
test criterium: 5.03259
****CLOSEST DISTRIBUTION INFO*** END
Watchdog operation finished, all ok.
```

Obrázek 7.1: Testovací data - vyhodnocení

8 Benchmark

Program jsem testoval na zařízení s následující konfigurací:

- int. grafická karta: Intel HD Graphics 530
- CPU: Intel Core i5-6500T @ 2.50 GHz
- RAM: 8 GB DDR4
- disk: 250 GB SSD

Testován byl soubor o velikosti 6,3GB reprezentující Gaussovo rozdělení. Dosáhl jsem následující doby běhu:

- SMP - 39,42s
- OpenCL, int. grafická karta - 51,62s
- ALL - 42,31s

Výsledek při využití CPU i OpenCL grafické karty se nachází v rozmezí ostatních uvedených časů. Vzhledem k tomu, že se jedná o integrovanou grafickou kartu, mi přijde výsledek uspokojivý.

9 Závěr

Byl vytvořen software v jazyce C++ (C pro OpenCL zařízení), který umožňuje vyhodnotit, jakému rozdělení jsou vstupní data nejbližší. Aplikace byla otestována na dodaných vstupních datech, kde funguje spolehlivě a u všech 4 souborů detekuje odpovídající rozdělení.

Paralelní běh byl vyzkoušen i na datech, jež jsem vytvořil pomocí dodaného generátoru. Aplikace i v tomto případě během testů fungovala dle očekávání.