

Přesnost a rychlost výpočtu

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 12

BOB36PRP – Procedurální programování

Přehled témat

- Část 1 – Přesnost výpočtu
Přesnost výpočtů a numerická stability
- Část 2 – Rychlost výpočtu (programu)
Maticové násobení
Rychlost výpočtu
Comparing C to Machine Code
Paralelní výpočet
- Část 3 – Implementace domácích úkolů
Domácí úkol HW10B

Přesnost výpočtů

Část I

Část 1 – Přesnost výpočtu

Přesnost výpočtů

Přesnost výpočtu - Příklad součtu dvou čísel

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double a = 1e+10;
6     double b = 1e-10;
7
8     printf("a : %24.121f\n", a);
9     printf("b : %24.121f\n", b);
10    printf("a+b: %24.121f\n", a + b);
11
12    return 0;
13 }
14
15 clang sum.c && ./a.out
16 a : 10000000000.000000000000
17 b : 0.000000000000100
18 a+b: 10000000000.000000000000
```

lec12/sum.c

Přesnost výpočtů

Přesnost výpočtu - Příklad dělení dvou čísel

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     const int number = 100;
6     double dV = 0.0;
7     float fV = 0.0f;
8
9     for (int i = 0; i < number; ++i) {
10        dV += 1.0 / 10.0;
11        fV += 1.0 / 10.0;
12    }
13
14    printf("double value: %lf ", dV);
15    printf("float value: %lf ", fV);
16
17    return 0;
18 }
19
20 clang division.c && ./a.out
21 double value: 10.000000 float value: 10.000002
```

lec12/division.c

Přesnost výpočtů

Přesnost výpočtu - strojová přesnost

- Strojová přesnost ϵ_m - nejmenší desetinné číslo, které přičtením k 1.0 dává výsledek různý od 1, pro $|v| < \epsilon_m$, platí

$$v + 1.0 == 1.0.$$

Symbol `==` odpovídá porovnání dvou hodnot (test na ekvivalenci).

- Zaokrouhlovací chyba - nejméně ϵ_m .
- Přesnost výpočtu - aditivní chyba roste s počtem operací v řádu $\sqrt{N} \cdot \epsilon_m$.
 - Často se však kumuluje preferabilně v jedno směru v řádu $N \cdot \epsilon_m$.

Přesnost výpočtů

Zdroje a typy chyby

- Chyby matematického modelu - matematická aproximace fyzikální situace.
- Chyby vstupních dat.
- Chyby numerické metody.
- Chyby zaokrouhlovací.

- Absolutní chyba aproximace
 $E(x) = \hat{x} - x$, \hat{x} přesná hodnota, x aproximace.
- Relativní chyba $RE(x) = \frac{\hat{x} - x}{x}$.

Přesnost výpočtů

Podmíněnost numerických úloh

- Podmíněnost úlohy $C_p = \frac{\text{relativní chyba výstupních údajů}}{\text{relativní chyba vstupních údajů}}$.
- Dobře podmíněná úloha $C_p \approx 1$.
- Výpočet je dobře podmíněný, je-li málo citlivý na poruchy ve vstupních datech.
- Numericky stabilní výpočet - vliv zaokrouhlovacích chyb na výsledek je malý.
- Výpočet je stabilní, je-li dobře podmíněný a numericky stabilní.

Přesnost výpočtů

Možnosti zvýšení přesnosti

- Reprezentace racionálních čísel - podíl dvou celočíselných hodnot, např. *Homogenní souřadnice*.
- „Libovolná přesnost“ - speciální knihovny, např. `gmp` až do výše volné paměti.
<https://gmplib.org/manual/index>
 - Souřadnice x, y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.

Přesnost výpočtu

Součin dvou velkých čísel knihovnou gmp - 1/2

- V HW04B je uveden příklad (995663 - 995669)⁸ jako prvočíselný rozklad čísla
932865073719992059629773513614789388266580305083920591925740371392254317064584855785088915745761.
<https://cv.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw04>
- Použijme knihovnu gmp pro mocninu a součin dvou čísel, #include<gmp.h>.
 - Typ celých čísel mpz_t, pomocné funkce mpz_init_set_str(), mpz_init(), gmp_printf() a mpz_clears() a operace mpz_pow_ui() a mpz_mul().
- Mocnina unsigned integer a násobení - multiplication. Knihovna nemusí být součástí operačního systému, proto může být nutně specifikovat cestu k hlavičkovému souboru a vlastní knihovně (-lgmp).
 - Můžeme zadat cestu ručně při kompilaci (nebo do Makefile).
 - Alternativně můžeme použít nástroj pkg-config (nebo pkgconf).
- Argumenty pro překlad (CFLAGS).
 - Argumenty pro linkování (LDFLAGS).

```
$ pkgconf --cflags gmp
-I/usr/local/include

$ pkgconf --libs gmp
-L/usr/local/lib -lgmp
```

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 11 / 43

Přesnost výpočtu

Součin dvou velkých čísel knihovnou gmp - 2/2

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <gmp.h>
5
6 const char* resultSrc =
7 "932865073719992059629773513614789388266580305083"
8 "920591925740371392254317064584855785088915745761";
9
10 int main(int argc, char *argv[])
11 {
12     int ret = EXIT_SUCCESS;
13     mpz_t n1, n2, result;
14     mpz_init_set_str(n1, "995663", 10);
15     mpz_init_set_str(n2, "995669", 10);
16     mpz_init(result);
17
18     gmp_printf("n1: %Zd\n", n1);
19     gmp_printf("n2: %Zd\n", n2);
20
21     gmp_printf("%Zd * %Zd = %Zd\n", n1, n2, result);
22
23     mpz_pow_ui(n1, n1, 8);
24     mpz_pow_ui(n2, n2, 8);
25 }
```

```
./demo-gmp-mpz
n1: 995663
n2: 995669
995663^8 x 995669^8
965826124294607867982699926255695296863400309121 x
96587268686826115153703708226023156648104775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
Result from HW04
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
```

lec12/gmp/demo-gmp-mpz.c

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 12 / 43

Přesnost výpočtu

Racionální čísla knihovny gmp - 1/3

- „Libovolné přesnosti“ reprezentace, např. souřadnic v rovině jako výsledek operací výpočetní geometrie, můžeme realizovat podílem dvou („libovolně velkých“) celých čísel.
 - Souřadnice x,y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.
- Knihovna gmp k tomuto účelu poskytuje typ mpq_t, kromě typu necelého čísla mpf_t, který využijeme pro převod mpq_t na celé číslo typu double.

```
1 double mpq2d(const mpq_t *op)
2 {
3     double ret;
4     mpf_t v;
5     mpf_init(v);
6     mpf_set_q(v, *op);
7     ret = mpf_get_d(v);
8     mpf_clear(v);
9     return ret;
10 }
```

lec12/gmp/demo-gmp-mpq.c

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 13 / 43

Přesnost výpočtu

Racionální čísla knihovny gmp - 2/3

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <gmp.h>
5
6 double mpq2d(const mpq_t *op);
7
8 int main(int argc, char *argv[])
9 {
10     int ret = EXIT_SUCCESS;
11
12     unsigned long xl = 75111641767681;
13     unsigned long yl = 3468686699521;
14     unsigned long denl = 37395671041;
15     const unsigned int digits = 21;
16
17     double xd = 1. * xl;
18     double yd = 1. * yl;
19     double dend = 1. * denl;
20
21     printf("unsigned long: %Lu %Lu %Lu\n", xl, yl, denl);
22     printf("double: %f %f %f\n", xd, yd, dend);
23
24     printf("double x,y (.2): %f %f\n", xd/dend, yd/dend);
25     printf("double x,y (.46): %f %f\n", xd/dend, yd/dend);
26 }
```

```
30 mpq_t x, y;
31 mpq_inits(x, y, NULL);
32 mpq_set_ui(x, xl, denl);
33 mpq_set_ui(y, yl, denl);
34
35 mpq_canonicalize(x);
36 mpq_canonicalize(y);
37
38 mpf_t xmpf, ympf;
39 mpf_inits(xmpf, ympf, NULL);
40 mpf_set_q(xmpf, x);
41 mpf_set_q(ympf, y);
42
43 gmp_printf("mpq x,y (canonical form): %04d %04d\n", x, y);
44 gmp_printf("mpf x,y (to %d decimal digits): %.*f %.*f\n",
45            digits, digits, xmpf, digits, ympf);
46 gmp_printf("mpq x,y (double .46): %f %f\n",
47            mpq2d(x), mpq2d(y));
48
49 mpq_clears(x, y, NULL);
50 mpf_clears(xmpf, ympf, NULL);
51 return ret;
```

lec12/gmp/demo-gmp-mpq.c

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 14 / 43

Přesnost výpočtu

Racionální čísla knihovny gmp - 3/3

- Souřadnice x,y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.

```
$ make
clang -c -I/usr/local/include -g demo-gmp-mpq.c -o demo-gmp-mpq.o
clang demo-gmp-mpq.o -I/usr/local/lib -lgmp -o demo-gmp-mpq
clang -c -I/usr/local/include -g demo-gmp-mpz.c -o demo-gmp-mpz.o
clang demo-gmp-mpz.o -I/usr/local/lib -lgmp -o demo-gmp-mpz

$ ./demo-gmp-mpq
unsigned long: 7511164176768 346868669952 3739567104
double: 7511164176768 346868669952 3739567104
double x,y (.2): 2008.57 92.76
double x,y (.46): 2008.5651541681761500512948295711265563964843750000
92.7563700036227487544238101691007614135742187500

mpq x,y (canonical form): 399190273/198744 1536231/16562
mpf x,y (to 21 decimal digits): 2008.565154168176146200000 92.756370003622750875500
mpq x,y (double .46): 2008.5651541681759226776193827390670776367187500000
92.7563700036227487544238101691007614135742187500
```

lec12/gmp/demo-gmp-mpq.c

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 15 / 43

Přesnost výpočtu

Makefile s pkg-config a gmp

```
1 CFLAGS+=$(shell pkg-config --cflags gmp)
2 LDFLAGS+=$(shell pkg-config --libs gmp)
3
4 CFLAGS+=-g
5
6 DEMO_MPQ=demo-gmp-mpq
7 DEMO_MPZ=demo-gmp-mpz
8
9 TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ)
10
11 bin: $(TARGETS)
12
13 info:
14 @echo $(CFLAGS)
15 @echo $(LDFLAGS)
```

```
17 $(DEMO_MPQ): $(DEMO_MPQ).o
18 $(CC) $(CFLAGS) $(LDFLAGS) -o $@
19
20 $(DEMO_MPZ): $(DEMO_MPZ).o
21 $(CC) $(CFLAGS) $(LDFLAGS) -o $@
22
23 %.o: %.c
24 $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
25
26 clean:
27 $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *
```

```
$ make info
-I/usr/local/include -g
-L/usr/local/lib -lgmp
```

```
$ gmake
clang -c -I/usr/local/include -g demo-gmp-mpq.c -o demo-gmp-mpq.o
clang demo-gmp-mpq.o -I/usr/local/lib -lgmp -o demo-gmp-mpq
clang -c -I/usr/local/include -g demo-gmp-mpz.c -o demo-gmp-mpz.o
clang demo-gmp-mpz.o -I/usr/local/lib -lgmp -o demo-gmp-mpz
```

lec12/gmp/Makefile

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 16 / 43

Přesnost výpočtu

Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru $x = (-1)^s \cdot \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$. Základ 2. IEEE 754, ISO/IEC/IEEE 60559:2011
- Mantisa je normalizována na první jedničku vlevo (v soustavě o dvojkovém základu).
- float – 32 bitů (4 bajty): s – 1 bit znaménko (+ nebo -), exponent – 8 bitů, tj. 256 možností. mantisa – 23 bitů ≈ 16,7 milionu možností.

- double – 64 bitů (8 bajtů).
 - s – 1 bit znaménko (+ nebo -).
 - exponent – 11 bitů, tj. 2048 možností.
 - mantisa – 52 bitů ≈ 4,5 biliónů možností.
- bias umožňuje reprezentovat exponent vždy jako kladné číslo. Lze zvolit, např. bias = 2^{eb-1} - 1, kde eb je počet bitů exponentu. <http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tecky>

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 17 / 43

Přesnost výpočtu

Příklad reprezentace float hodnot dle IEEE 754

- Chyba reprezentace -256.75 vs -256.74.
- Infinity (0x7f800000), -Infinity (0xf8000000), a NaN (0x7fffffff).

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 18 / 43

Přesnost výpočtu

Příklady reprezentace hodnot typu float

- Reprezentace čísla 85,125 (float)
 - 85 odpovídá 1010101₍₂₎.
 - 0,125 odpovídá 001
 - 0,125/2⁻¹ = 0,25 | 0
 - 0,125/2⁻² = 0,50 | 0
 - 0,125/2⁻³ = 1,00 | 1
 - 85,125 odpovídá 1010101,001₍₂₎ = 1,010101001₍₂₎ × 2⁸.
 - Bias pro float je 127.
 - Exponent je 127 + 6 = 133
 - 133 odpovídá 10000101₍₂₎.
 - Normalizovaná mantisa je 010101001₍₂₎, kterou doplníme nulami na 23 bitů (zprava, je to desetinné číslo).
 - 0-1000 0101-0101 0100 1000 0000 0000 0000.
 - 01000010 10101010 01000000 00000000.
 - V šestnáctkové soustavě 0x42 0xaa 0x40 0x00, tedy 0x42aa4000.
- Reprezentace čísla 0,1 (float)
 - 0,1 má periodický rozvoj
 - 0,1 * 2 = 0,2 | 0
 - 0,2 * 2 = 0,4 | 0
 - 0,4 * 2 = 0,8 | 0
 - 0,8 * 2 = 1,6 | 1
 - 0,6 * 2 = 1,2 | 1
 - 0,2 * 2 = 0,4 | 0
 - Opakuje se 0011, 23-bitů tak reprezentuje menší hodnotu.
 - 0,1₍₁₀₎ ~ 0,0001 1001 1001 1001 1001 1001 100₍₂₎ = 1,1001 1001 1001 1001 1001 100₍₂₎ × 2⁻⁴.
 - Exponent je 127 - 4 = 123 odpovídá 0111 1011₍₂₎.
 - Normalizovaná mantisa 11001100 1100 1100 1100 1100 1100.
 - 0-0111 1011-100 1100 1100 1100 1100 1100.
 - 00111011 11001100 11001100 11001100.
 - V šestnáctkové soustavě to je 0x3d 0xcc 0xcc 0xcc, tedy 0x3dcccccc.
 - Prakticky je 0,1 převedeno na o něco větší číslo 0x3dcccccd, protože absolutní chyba je menší.

lec12/floats.c

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přesnost a rychlost výpočtu 19 / 43

Sčítání mnoha malých necelých čísel - 1/2

- Na příkladu součtu dvou velmi odlišných čísel (např. $1 \times 10^{10} + 1 \times 10^{-10}$) dochází z důvodu omezené reprezentace mantisy k zaokrouhlovací chybě.
- V případě naivní implementace součtu velkého počtu (např. 2^{30}) velmi malých hodnot (např. 1×10^{-20}) může dojít vlivem zaokrouhlování k významné chybě.

lec12/addition.c

```
// small value to be sum
float v = 1e-20f; //float literal
// 1073741824 is 2^30 values (1e9)
const size_t power = 30;
size_t n = 1l<<power;

// multiplication factor for print
const double k = 1e11;

float sum_naive(size_t n, float v)
{
    float r = 0;
    for (size_t i = 0; i < n; ++i) {
        r += v[i];
    }
    return r;
}

float sum_alter(size_t n, float v, size_t power)
{
    float r = 0;
    const size_t order = power - 1;
    size_t k = 2;
    for (size_t l = 1; l < order; ++l, k *= 2) {
        for (size_t i = 0; i < n; i += k) {
            v[i] = v[i] + v[i+k/2];
        }
    }
    k /= 2;
    for (size_t i = 0; i < n; i += k) {
        r += v[i];
    }
    return r;
}

double sum1 = v*n * k;
double sum2 = sum_naive(n, values) * k;
float sum3 = sum_alter(n, values, power) * k;

Přímé násobení - výsledek 1.0737417899253642872281
Naivní součet - výsledek 0.0227373675443232059479
Sčítání po dvojicích - výsledek 1.0737417936325073242188
```

Sčítání mnoha malých necelých čísel - 2/2

```
#include <stdio.h>
#include <stdlib.h>

float* init_values(size_t n, float v);
float sum_naive(size_t n, float v);
float sum_alter(size_t n, float v, size_t power);

int main(void)
{
    float v = 1e-20f; // small value to be sum
    const size_t power = 30; // try 3 vs. 30
    size_t n = 1l<<power; // 1073741824 is 2^30 values
    const double k = 1e11;

    float *values = init_values(n, v);
    double sum1 = v * n * k;
    double sum2 = sum_naive(n, values) * k;
    float sum3 = sum_alter(n, values, power) * k;

    printf("Sum of %lu numbers of the value %.22f\n", n, v);
    printf("Sum1 (multiplication): %.22f\n", sum1);
    printf("Sum2 (naive) : %.22f\n", sum2);
    printf("Sum3 (alter) : %.22f\n", sum3);
    free(values);
    return EXIT_SUCCESS;
}

float* init_values(size_t n, float v)
{
    float *r = malloc(n * sizeof(v));
    if (!r) {
        fprintf(stderr, "ERROR: MEM_ALLOCA\n");
        exit(-1);
    }
    for (size_t i = 0; i < n; ++i) {
        r[i] = v;
    }
    return r;
}

$ clang addition.c -o addition && ./addition
Sum of 1073741824 numbers of the value
0.00000000000000000000000000000000
Sum1 (multiplication): 1.0737417899253642872281
Sum2 (naive) : 0.0227373675443232059479
Sum3 (alter) : 1.0737417936325073242188

$ calc "1e-20 * 2^30 * 1e11"
1.073741824

lec12/addition.c
Implementujte s využitím knihovny gmp.
```

Část II

Část 2 – Rychlost výpočtu (programu)

Maticové násobení - Naivně

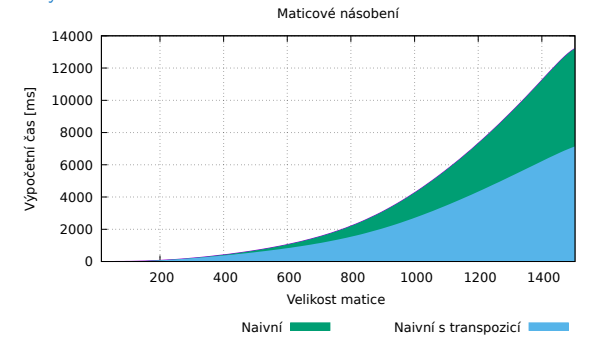
```
void simple_multiply(const int n, const double *a, const double *b, double *c)
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double prod = 0;
            for (int k = 0; k < n; ++k) {
                prod += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = prod;
        }
    }
}
```

- Pro přehlednost předpokládáme kompatibilní rozměry matic a správně alokované.

Maticové násobení - Naivně s transpozicí

```
void simple_multiply_trans(const int n, const double *a, const double *b, double *c)
{
    double *bT = create_matrix(n); // allocate memory for transposed matrix
    for (int i = 0; i < n; ++i) {
        bT[i*n + i] = b[i*n + i];
        for (int j = i + 1; j < n; ++j) {
            bT[i*n + j] = b[j*n + i];
            bT[j*n + i] = b[i*n + j];
        }
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double tmp = 0;
            for (int k = 0; k < n; ++k) {
                tmp += a[i*n + k] * bT[j*n + k];
            }
            c[i*n + j] = tmp;
        }
    }
    free(bT);
}
```

Porovnání rychlosti násobení matic



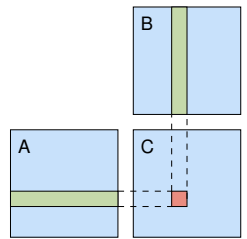
Architektura procesoru a způsob výpočtu

- Příklad násobení matic a násobení transponované matice ukazuje, že kromě instrukcí má zásadní vliv organizace dat a přístup do paměti.

- V moderních procesorech hraje cache zásadní roli společně s řetězením instrukcí, tzv. pipelining, a využitím specifických instrukcí.

SIMD - Single Instruction Multiple Data

- Proniknutí do detailů fungování cache a řetězení instrukcí je náplní předmětu **Architektura počítačů (BOB35APO)**, kde máte možnost se seznámit s přicházející architekturou RISC V.



<https://cw.felk.cvut.cz/wiki/courses/b35apo/>

Optimalizace kódu

- Kromě optimalizace výsledného spustitelného kódu při překladu, je možné optimalizovat kódu za běhu nebo již existujících binární (přeložené) soubory.

- BOLT** - Binary Optimization and Layout Tool, zrychlení o až 20%–50% <https://arxiv.org/abs/1807.06735>
- <https://dl.acm.org/doi/10.5555/3314872.3314876>

- Využití speciálních instrukcí v základních funkcích může výpočty (programy) výrazně urychlit, zejména pokud se funkce používají masivně.

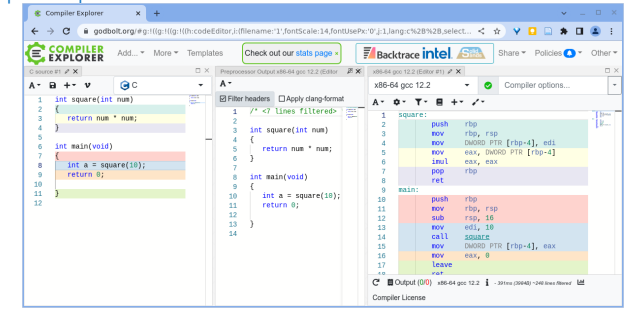
- AVX2** a **EVEX** instrukce (ze sady SSE4.2) ve funkcích porovnání řetězců `str{n}.casecmp()` – až o 38% méně potřebného času.

03/2022 - <https://www.phoronix.com/news/Glibc-strcasecmp-AVX2-EVEX>

- V obou případech (a obecně) je vhodné rozumět principu a využít instrukce Assembleru.*

Informativní

Compiler Explorer



<https://godbolt.org/z/K9r1eWqd>

Maticové násobení Rychlost výpočtu Comparing C to Machine Code Paralelní výpočet

Compiler Explorer – Analýza optimalizovaného kódu

■ Vliv optimalizace -O2 na výsledný kód, který obsahuje nedefinované chování, přetečení celého císla. Příloha 3. přednášky.

<https://godbolt.org/z/G3GEz4vbv>

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 31 / 43

Maticové násobení Rychlost výpočtu Comparing C to Machine Code Paralelní výpočet

Comparing C to Machine Code

<https://www.youtube.com/watch?v=y0yaJXpAYZQ>

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 33 / 43

Maticové násobení Rychlost výpočtu Comparing C to Machine Code Paralelní výpočet

Příklad použití OpenMP - Maticové násobení 1/2

■ Open Multi-Processing (OpenMP) - aplikační programové rozhraní (API) multiplatformních výpočtů se sdílenou pamětí. <http://www.openmp.org>

■ Direktivu preprocessoru můžeme instruovat kompilátor k vytvoření kódu paralelního výpočtu, např. paralelizace přes vnější proměnnou i.

```

1 void multiply(int n, int a[n][n], int b[n][n], int c[n][n])
2 {
3     int i;
4     #pragma omp parallel private(i)
5     #pragma omp for schedule (dynamic, 1)
6     for (i = 0; i < n; ++i) {
7         for (int j = 0; j < n; ++j) {
8             c[i][j] = 0;
9             for (int k = 0; k < n; ++k) {
10                c[i][j] += a[i][k] * b[k][j];
11            }
12        }
13    }
14 }

```

`lec12/demo-omp-matrix.c`

Pro přehlednost uvažujeme čtvercové matice stejných rozměrů.

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 35 / 43

Maticové násobení Rychlost výpočtu Comparing C to Machine Code Paralelní výpočet

Příklad použití OpenMP - Maticové násobení 2/2

■ Příklad násobení matic 1000 x 1000 s využitím OpenMP na iCore5 (2 jádra s HT ~ 4x výpočetní jednotky).

■ Násobení matic 5000 x 5000 (Ryzen 9 5950X).

```

1 gcc -std=c99 -O2 -o demo-omp demo-omp-matrix.c -fopenmp
2 ./demo-omp 1000
3 Size of matrices 1000 x 1000 naive
4 multiplication with 0(n^3)
5 c1 == c2: 1
6 Multiplication single core 9.33 sec
7 Multiplication multi-core 4.73 sec
8
9 OMP_NUM_THREADS=2 ./demo-omp 1000
10 Size of matrices 1000 x 1000 naive
11 multiplication with 0(n^3)
12 c1 == c2: 1
13 Multiplication single core 9.48 sec
14 Multiplication multi-core 6.23 sec

```

```

1 OMP_NUM_THREADS=16 ./demo-omp 5000
2 Size of matrices 5000 x 5000 naive
3 multiplication with 0(n^3)
4 Multiplication single core 256.00 sec
5 Multiplication multi-core 18.05 sec

```

`lec12/demo-omp-matrix.c`

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 36 / 43

Domácí úkol HW10B

Část III

Část 3 – Implementace domácích úkolů (HW10B)

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 37 / 43

Domácí úkol HW10B

Zrychlení domácího úkolu HW10B

■ V případě správného použití prioritní fronty haldou, je nejvíce času programu stráveno

- načítáním grafu z textového souboru a
- ukládáním výsledku do textového souboru.

■ V obou případech se jedná tři celá čísla (v rozsahu `int`) na řádek.

- Prevádíme znaky na celá čísla a zpět.
- Referenční řešení (`ref-lec12`) využívá funkce `fscanf()` a `fprintf()`, které jsou relativně komplexní a pomalé.

Legend

- Load time: ■ ref-lec12 - reference implementation provided as a part of the source codes for the lecture 12.
- Solve time: ■ rdijkstra - baseline reference solution.
- Save time: ■ ref-mh21 - reference solution, the fastest student implementation of the year 2019 (updated to 2021 standards).
- ref-radix-mh21 - reference solution with a variant of radix queue and even faster save (student's implementation updated to 2021 standards).

Implementation	Load time (ms)	Solve time (ms)	Save time (ms)
ref-mh21	5547	6590	11419
ref-radix-mh21	5547	6590	11419
rdijkstra	5547	6590	11419
ref-lec12	5547	6590	11419

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 39 / 43

Domácí úkol HW10B

Zrychlení domácího úkolu HW10B - Načítání

■ Při načítání grafu můžeme využít faktu, že vstupní soubor obsahuje pouze kladná čísla oddělená mezerami v rozsahu `int`.

■ Načítání můžeme urychlit přímou konverzí načteného znaku na celé číslo.

■ Princip načtení celého čísla z řetězce může být následující. *Vstup je posloupnost znaků.*

```

1 int parse_int(const char *str, int len)
2 {
3     int num = 0;
4     int i = 0;
5     while (i < len && str[i] >= '0' && str[i] <= '9') {
6         num = num * 10 + (str[i++] - '0');
7     }
8     return str[i] == ' ' || str[i] == '\n' ? num : -1;
9 }

```

`lec12/int_string.c`

Vyžadujeme oddělení číselných hodnot mezerou nebo znakem nového řádku a předpokládáme `str[i] == '\0'`.

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 40 / 43

Domácí úkol HW10B

Zrychlení domácího úkolu HW10B - Ukládání

■ Můžeme využít maximálního počtu znaků v rozsahu typu `int` a výstupní posloupnost znaků reprezentující celé číslo vytvářet od nejnižšího řádu postupným dělením.

■ Princip konverze kladného celého čísla může být následující. *Výstup je posloupnost znaků.*

```

1 char* int_to_string(int v, char *buf, size_t capacity)
2 {
3     char *cur = buf + capacity - 1; //last char of the buffer buf
4     *cur = '\0';
5     do {
6         int least = v % 10;
7         v /= 10;
8         * (--cur) = least + '0';
9     } while (v != 0);
10    return cur;
11 }

```

`lec12/int_string.c`

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 41 / 43

Diskutovaná témata

Shrnutí přednášky

Jan Faigl, 2023 BOB36PRP – Přednáška 12: Přenos a rychlost výpočtu 42 / 43

Diskutovaná témata

- Numerická přesnost.
- Knihovna `gmp`.
- Maticové násobení a organizace paměti.
- Rychlost výpočtu a architektura procesoru.
- Paralelní výpočty `OpenMP`.
- Domácí úkol HW10B.