

# Přesnost a rychlost výpočtu

Jan Faigl

Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

Lecture 10

PRG – Programming in C

## Přehled témat

- Část 1 – Přesnost výpočtu  
Přesnost výpočtů a numerická stability
- Část 2 – Rychlost výpočtu (programu)  
Maticové násobení  
Rychlost výpočtu  
Paralelní výpočet
- Část 3 – Implementace domácích úkolů  
Diskutovaná témata

Přesnost výpočtů

Part 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 : 100000000000.000000000000000000
17 b : 0.000000000000000000000000000000
18 a+b: 100000000000.000000000000000000
```

lec10/sum.c

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

## 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
```

lec10/division.c

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

## Přesnost výpočtu - strojová přesnost

- Strojová přesnost  $\epsilon_m$  - 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  $\epsilon_m$ .
- 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

## 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 výpočtů

### Součin dvou velkých čísel knihovnou gmp - 1/2

- V HW04B je uveden příklad (995663 - 995669)<sup>8</sup> 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, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 11 / 36

Přesnost výpočtů

### Součin dvou velkých čísel knihovnou gmp - 2/2

```
1 #include <stdio.h> 26 gmp_printf("Xz x Xz\n", n1, n2);
2 #include <stdlib.h> 27
3 28 mpz_mul(result, n1, n2);
4 #include <gmp.h> 29 gmp_printf("Xz\n", result);
5 30
6 const char* resultSrc = 31 printf("Result from HW04\nXz\n", resultSrc);
7 "932865073719992059629773513614789388266580305083" 32
8 "920591925740371392254317064584855785088915745761"; 33
9 34 mpz_clears(n1, n2, result, NULL);
10 35 return ret;
11 36 }
12 int main(int argc, char *argv[]) 37
13 { 38
14 int ret = EXIT_SUCCESS; 39 $ ./demo-gmp-mpz
15 mpz_t n1, n2, result; 40 n1: 995663
16 mpz_init_set_str(n1, "995663", 10); 41 n2: 995669
17 mpz_init_set_str(n2, "995669", 10); 42 995663*8 x 995669*8
18 mpz_init(result); 43 96582612429460787982699926255695296863400309121 x
19 gmp_printf("n1: Xz\n", n1); 44 96587268868626115153703708226023156648104775841
20 gmp_printf("n2: Xz\n", n2); 45 93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
21 gmp_printf("Xz Xz x Xz Xz\n", n1, 8, n2, 8); 46 Result from HW04
22 47 93286507371999205962977351361478938826658030508392059
23 mpz_pow_ui(n1, n1, 8); 48 1925740371392254317064584855785088915745761
24 mpz_pow_ui(n2, n2, 8); 49 96582612429460787982699926255695296863400309121 x
96587268868626115153703708226023156648104775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
50 51 double mpq2d(const mpq_t *op)
52 {
53 double ret;
54 mpf_t v;
55 mpf_init(v);
56 mpf_set_q(v, *op);
57 ret = mpf_get_d(v);
58 mpf_clear(v);
59 return ret;
60 }
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 12 / 36

Přesnost výpočtů

### Racionální čísla knihovny gmp - 1/3

- „Libovolné přesnosti“ reprezentace, např. souřadnic v rovině jako výsledek operací výpočetní geometrie, můžeme realizovat podílem dvou („libovolné 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.

```
49 double mpq2d(const mpq_t *op)
50 {
51 double ret;
52 mpf_t v;
53 mpf_init(v);
54 mpf_set_q(v, *op);
55 ret = mpf_get_d(v);
56 mpf_clear(v);
57 return ret;
58 }
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 13 / 36

Přesnost výpočtů

### Racionální čísla knihovny gmp - 2/3

```
1 #include <stdio.h> 27 mpq_t x, y;
2 #include <stdlib.h> 28 mpq_inits(x, y, NULL);
3 #include <gmp.h> 29 mpq_set_ui(x, x1, den1);
4 30 mpq_set_ui(y, y1, den1);
5 double mpq2d(const mpq_t *op); 31
6 32 mpq_canonicalize(x);
8 int main(int argc, char *argv[]) 33 mpq_canonicalize(y);
9 { 34
10 int ret = EXIT_SUCCESS; 35 mpf_t xmpf, ympf;
11 36 mpf_inits(xmpf, ympf, NULL);
12 unsigned long x1 = 75111641767681; 37 mpf_set_q(xmpf, x);
13 unsigned long y1 = 3468686699521; 38 mpf_set_q(ympf, y);
14 unsigned long den1 = 37395671041; 39
15 const unsigned int digits = 21; 40 gmp_printf("mpq x,y (canonical form): X04 X04\n", x, y);
16 41 gmp_printf("mpf x,y (to Xd decimal digits): %.Xf %.Xf\n",
17 double xd = 1. * x1; 42 mpf_printf("mpq x,y (double .46): %.46lf %.46lf\n",
18 double yd = 1. * y1; 43 mpq2d(x), mpq2d(y));
19 double dennd = 1. * den1; 44
20 45 mpq_clears(x, y, NULL);
21 printf("unsigned long: X1u X1u\n", x1, y1, den1); 46 mpf_clears(xmpf, ympf, NULL);
22 printf("double: %.X0lf %.X0lf\n", xd, yd, dennd); 47 return ret;
23 48
24 printf("double x,y (.2): %.2lf %.2lf\n", xd/dennd, yd/dennd);
25 printf("double x,y (.46): %.46lf %.46lf\n", xd/dennd, yd/dennd);
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 14 / 36

Přesnost výpočtů

### 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 -L/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
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 15 / 36

Přesnost výpočtů

### Makefile s pkg-config a gmp

```
1 CFLAGS+=$(shell pkg-config --cflags gmp) 17 $(DEMO_MPQ): $(DEMO_MPQ).o
2 LDFLAGS+=$(shell pkg-config --libs gmp) 18 $(CC) $(C) $(LDFLAGS) -o $@
3 19
4 CFLAGS+=-g 20 $(DEMO_MPZ): $(DEMO_MPZ).o
5 DEMO_MPQ=demo-gmp-mpq 21 $(CC) $(C) $(LDFLAGS) -o $@
6 DEMO_MPZ=demo-gmp-mpz 22
7 DEMO_MPZ=demo-gmp-mpz 23 %.o : %.c
8 TARGETS+=$(DEMO_MPQ) $(DEMO_MPZ) 24 $(CC) -c $(CPPFLAGS) $(CFLAGS) $(C) -o $@
9 25
10 26 clean:
11 bin: $(TARGETS) 27 $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o
12 13 info:
14 @echo $(CFLAGS)
15 @echo $(LDFLAGS)
```

```
$ 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 -L/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 -L/usr/local/lib -lgmp -o demo-gmp-mpz
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 16 / 36

Přesnost výpočtů

### Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru  $x = (-1)^s \cdot \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$ . 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 bilióny možností (4 503 599 627 370 495).
- bias umožňuje reprezentovat exponent vždy jako kladné číslo.
  - Lze zvolit, např. bias = 2<sup>eb-1</sup> - 1, kde eb je počet bitů exponentu.

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 17 / 36

Přesnost výpočtů

### Příklad reprezentace float hodnot dle IEEE 754

- Chyba reprezentace -256.75 vs -256.74.
- Infinity (0x7f800000), -Infinity (0xff800000), a NaN (0xf7ffffff).

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 18 / 36

Přesnost výpočtů

### Příklady reprezentace hodnot typu float

- Reprezentace čísla 85,125 (float)
  - 85 odpovídá 1010101(2).
  - 0,125 odpovídá 001
    - 0,125/2<sup>-1</sup> = 0,25 | 0
    - 0,125/2<sup>-2</sup> = 0,50 | 0
    - 0,125/2<sup>-3</sup> = 1,00 | 1
  - 85,125 odpovídá 1010101,001(2) = 1,010101001(2) × 2<sup>8</sup>.
  - 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 to je 0x3d 0xxx 0xxx 0xxx, 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<sub>(10)</sub> ~ 0,0001 1001 1001 1001 1001 1001 100(2) = 1,0011001100110011001100(2) × 2<sup>-4</sup>.
  - Exponent je 127 - 4 = 123 odpovídá 01111011(2).
  - Normalizovaná mantisa 10011001100110011001100.
  - 0-0111 1011-100110 11001100 11001100 1100.
  - 00111011 11001100 11001100 11001100.
  - V šestnáctkové soustavě to je 0x3d 0xxx 0xxx 0xxx, tedy 0x3dcccc.
  - Prakticky je 0,1 převedeno na o něco větší číslo 0x3dcccc, protože absolutní chyba je menší.

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 19 / 36

**Přesnost výpočtů**

### 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.  $1 \times 10^{-20}$ ) může dojít vlivem zaokrouhlování k významné chybě.

```
lec10/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 *values = init_values(n, v);

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

printf("Přímé násobení - výsledek\n1.0737417899253642872281.");
printf("Naivní součet - výsledek\n0.0227373675443232059479.");
printf("Sčítání po dvojicích - výsledek\n1.0737417936325073242188.");
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 20 / 36

**Přesnost výpočtů**

### Sčítání mnoha malých necelých čísel - 2/2

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float* init_values(size_t n, float v);
5 float sum_naive(size_t n, float v);
6 float sum_alter(size_t n, float v, size_t power);
7
8 int main(void)
9 {
10 float v = 1e-20f; // small value to be sum
11 const size_t power = 30; // try 3 vs. 30
12 size_t n = 1l<<power; // 1073741824 is 2^30 values
13 const double k = 1e11;
14
15 float *values = init_values(n, v);
16
17 double sum1 = v * n * k;
18 double sum2 = sum_naive(n, values) * k;
19 float sum3 = sum_alter(n, values, power) * k;
20
21 printf("Sum of %lu numbers of the value %.22f\n", n, v);
22 printf("Sum1 (multiplication): %.22f\n", sum1);
23 printf("Sum2 (naive) : %.22f\n", sum2);
24 printf("Sum3 (alter) : %.22f\n", sum3);
25 free(values);
26 return EXIT_SUCCESS;
27 }
```

```
29 float* init_values(size_t n, float v)
30 {
31 float *r = malloc(n * sizeof(v));
32 if (!r) {
33 fprintf(stderr, "ERROR: MEM_ALLOC\n");
34 exit(-1);
35 }
36 for (size_t i = 0; i < n; ++i) {
37 r[i] = v;
38 }
39 return r;
40 }
```

```
$ clang addition.c -o addition && ./addition
Sum of 1073741824 numbers of the value
0.00000000000000000000000100
Sum1 (multiplication): 1.0737417899253642872281
Sum2 (naive) : 0.0227373675443232059479
Sum3 (alter) : 1.0737417936325073242188

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

lec10/addition.c

- Implementujte s využitím knihovny `gmp`.

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 21 / 36

Maticové násobení Rychlost výpočtu Paralelní výpočet

## Part II

### Část 2 – Rychlost výpočtu (programu)

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 22 / 36

**Maticové násobení - Naivně**

```
1 void simple_multiply(const int n, const double *a, const double *b, double *c)
2 {
3     for (int i = 0; i < n; ++i) {
4         for (int j = 0; j < n; ++j) {
5             double prod = 0;
6             for (int k = 0; k < n; ++k) {
7                 prod += a[i * n + k] * b[k * n + j];
8             }
9             c[i * n + j] = prod;
10        }
11    }
12 }
```

- Pro přehlednost předpokládáme kompatibilní rozměry matic a správně alokované.

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 24 / 36

**Maticové násobení - Naivně s transpozicí**

```
1 void simple_multiply_trans(const int n, const double *a, const double *b, double *c)
2 {
3     double *bT = create_matrix(n); // allocate memory for transposed matrix
4     for (int i = 0; i < n; ++i) {
5         bT[i*n + i] = b[i*n + i];
6         for (int j = i + 1; j < n; ++j) {
7             bT[i*n + j] = b[j*n + i];
8             bT[j*n + i] = b[i*n + j];
9         }
10    }
11    for (int i = 0; i < n; ++i) {
12        for (int j = 0; j < n; ++j) {
13            double tmp = 0;
14            for (int k = 0; k < n; ++k) {
15                tmp += a[i*n + k] * bT[j*n + k];
16            }
17            c[i*n + j] = tmp;
18        }
19    }
20    free(bT);
21 }
```

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 25 / 36

**Maticové násobení**

### Porovnání rychlosti násobení matic

Y-ová osa: Výpočetní čas [ms] (0 to 14000)  
X-ová osa: Velikost matice (200 to 1400)

Legenda: Naivní (zelená), Naivní s transpozicí (modrá)

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 26 / 36

**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/>

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 28 / 36

**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

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu 29 / 36

**Compiler Explorer**

<https://godbolt.org/z/K9r1eWqcd>

Jan Faigl, 2024 PRG – Lecture 10: Přesnost a rychlost výpočtu Informativní 30 / 36

Maticové násobení Rychlost výpočtu Paralelní výpočet

## Compiler Explorer – Analýza optimalizovaného kódu

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

<https://godbolt.org/z/G3GEz4vbw>

Jan Faigl, 2024 PRG – Lecture 10: Přenos a rychlost výpočtu 31 / 36

Maticové násobení Rychlost výpočtu 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>
- Direktivou preprocesoru 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 }

```

`lec10/demo-omp-matrix.c`  
Pro přehlednost uvažujeme čtvercové matice stejných rozměrů.

Jan Faigl, 2024 PRG – Lecture 10: Přenos a rychlost výpočtu 33 / 36

Maticové násobení Rychlost výpočtu 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 O(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 O(n^3)
12 c1 == c2: 1
13 Multiplication single core 9.48 sec
14 Multiplication multi-core 6.23 sec

```

- i7-6700K:
  - 1x vlákno 0.80s;
  - 2x vlákna 0.39s;
  - 4x vlákna 0.24s.

```

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

```

`lec10/demo-omp-matrix.c`

Jan Faigl, 2024 PRG – Lecture 10: Přenos a rychlost výpočtu 34 / 36

Diskutovaná témata

## Shrnutí přednášky

Jan Faigl, 2024 PRG – Lecture 10: Přenos a rychlost výpočtu 35 / 36

Diskutovaná témata

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

Jan Faigl, 2024 PRG – Lecture 10: Přenos a rychlost výpočtu 36 / 36