

Přesnost a rychlost výpočtu

Jan Faigl

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

Lecture 10

B0B36PRG – 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

Part I

Část 1 – Přesnost výpočtu

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 : 0.000000000100
18 a+b: 10000000000.000000000000
```

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 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 $N \cdot \epsilon_m$.

Zdroje a typy chyby

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

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

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.

Možnosti zvýšení přesnosti

- Reprezentace racionálních čísel - podíl dvou celočíselných hodnot, např. *Homogenní souřadnice*.
- „Libovolná přesnost“ - speciální knihovny, např. gmp až do výše volné paměti.
 - Souřadnice x,y - 7511164176768 346868669952 3739567104 ~ 2008,57; 92,76.

<https://gmplib.org/manual/index>

Součin dvou velkých čísel knihovnou gmp - 1/2

- V HW04B je uveden příklad $(995663 \cdot 995669)^8$ jako prvočíselný rozklad čísla 932865073719992059629773513614789388266580305083920591925740371392254317064584855785088915745761.
 - <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw02>
- 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 nutné 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`).
 - <https://www.freedesktop.org/wiki/Software/pkg-config/> <http://pkgconf.org/>
 - Argumenty pro překlad (`CFLAGS`).
 - Argumenty pro linkování (`LDFLAGS`).

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

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

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^%d x %Zd^%d\n", n1, 8, n2, 8);
22
23     mpz_pow_ui(n1, n1, 8);
24     mpz_pow_ui(n2, n2, 8);
25
26     gmp_printf("%Zd x %Zd\n", n1, n2);
27
28     mpz_mul(result, n1, n2);
29     gmp_printf("%Zd\n", result);
30
31     printf("Result from HW04\n", result);
32
33     mpz_clears(n1, n2, result, NULL);
34     return ret;
35 }
```

```

$ ./demo-gmp-mpz
n1: 995663
n2: 995669
995663^8 x 995669^8
965826124294607867982699926255695296863400309121 x
965872686868261151537037082260231566481047775841
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
Result from HW04
93286507371999205962977351361478938826658030508392059
1925740371392254317064584855785088915745761
lec10/gmp/demo-gmp-mpz.c
```

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 necelé 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 }
```

<lec10/gmp/demo-gmp-mpq.c>

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 = 7511641767681;
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: %.01f %.01f %.01f\n", xd, yd, dend);
23
24     printf("double x,y (.2): %.2lf %.2lf\n", xd/dend, yd/dend);
25     printf("double x,y (.46): %.46lf %.46lf\n", xd/dend, yd/dend);
26
27     mpq_t x, y;
28     mpq_inits(x, y, NULL);
29     mpq_set_ui(x, xl, denl);
30     mpq_set_ui(y, yl, denl);
31
32     mpq_canonicalize(x);
33     mpq_canonicalize(y);
34
35     mpf_t xmpf, ympf;
36     mpf_inits(xmpf, ympf, NULL);
37     mpf_set_q(xmpf, x);
38     mpf_set_q(ympf, y);
39
40     gmp_printf("mpq x,y (canonical form): %Qd %Qd\n", x, y);
41     gmp_printf("mpf x,y (to %d decimal digits): %.Ff %.Ff\n",
42               digits, digits, xmpf, ympf);
43     gmp_printf("mpq x,y (double .46): %.46lf %.46lf\n",
44               mpq2d(x), mpq2d(y));
45
46     mpq_clears(x, y, NULL);
47     mpf_clears(xmpf, ympf, NULL);
48     return ret;
49 }

```

lec10/gmp/demo-gmp-mpq.c

Racionální čísla knihovny gmp - 3/3

- Souřadnice x,y - 751164176768 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 -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

```

```

$ ./demo-gmp-mpq
unsigned long: 751164176768 346868669952 3739567104
double: 751164176768 346868669952 3739567104
double x,y (.2): 2008.57 92.76
double x,y (.46): 2008.5651541681761500512948259711265563964843750000
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
92.756370003622748754423810169100761413574218750

```

lec10/gmp/demo-gmp-mpq.c

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)
16
17 $(DEMO_MPQ): $(DEMO_MPQ).o
18     $(CC) $< $(LDFLAGS) -o $@
19
20 $(DEMO_MPZ): $(DEMO_MPZ).o
21     $(CC) $< $(LDFLAGS) -o $@
22
23 %.o : %.c
24     $(CC) -c $(CPPFLAGS) $(CFLAGS) $< -o $@
25
26 clean:
27     $(RM) $(DEMO_MPQ) $(DEMO_MPZ) *.o

```

lec10/gmp/Makefile

```

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

```

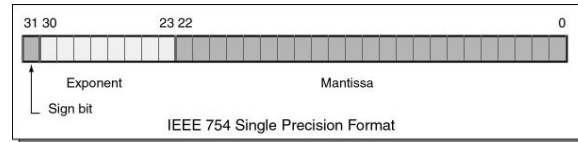
```

$ 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

```

Reprezentace necelých čísel – IEEE 754

- Reálné číslo x se zobrazuje ve tvaru $x = (-1)^s \text{mantisa} \cdot 2^{\text{exponent} - \text{bias}}$. 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 $\approx 16,7$ milionu možností.



- **double** – 64 bitů (8 bajtů).
 - s – 1 bit znaménko (+ nebo –).
 - **exponent** – 11 bitů, tj. 2048 možností.
 - **mantisa** – 52 bitů $\approx 4,5$ bilióny možností (4 503 599 627 370 495).
- **bias** umožňuje reprezentovat exponent vždy jako kladné číslo.

Lze zvolit, např. $\text{bias} = 2^{e-1} - 1$, kde e je počet bitů exponentu.
<http://www.root.cz/clanky/norma-ieee-754-a-pribuzni-formaty-plovouci-radove-tenky>

Part II

Část 2 – Rychlost výpočtu (programu)

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.

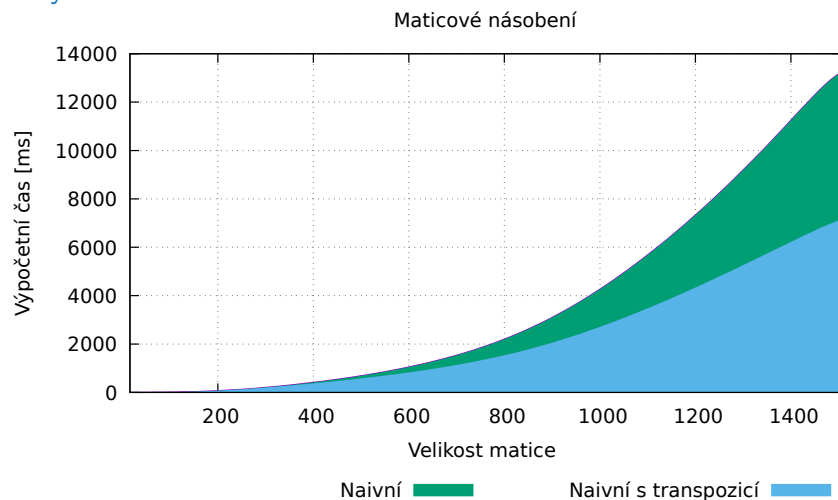
Maticové násobení - Naivně s transpozicí

```

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 }

```

Porovnání rychlosti násobení matic



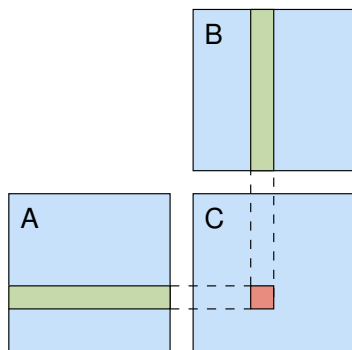
Architektura procesoru a způsob výpočtu

- 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 (BOB35AP0)**, 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 překládání, 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

```

1 int square(int num)
2 {
3     return num * num;
4 }
5
6 int main(void)
7 {
8     int a = square(10);
9     return 0;
10 }
11
12
13
14

```

```

1 square:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul  eax, eax
7     pop     rbp
8     ret
9
10 main:
11     push   rbp
12     mov   rbp, rsp
13     sub   rsp, 16
14     mov   edi, 10
15     call square
16     mov   DWORD PTR [rbp-4], eax
17     mov   eax, 0
18     leave
19     ret

```

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

Compiler Explorer – Analýza optimalizovaného kódu

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

Příloha 3. přednášky.

```

1 int main(void)
2 {
3     int ret = 0;
4     for (int i = 2147483640; i >= 0; ++i) {
5         ret += i;
6     }
7     return ret;
8 }

```

```

1 main:
2     push   rbp
3     mov   rbp, rsp
4     mov   DWORD PTR [rbp-4], 0
5     mov   DWORD PTR [rbp-8], 2147483640
6     jmp  .L2
7
8 .L3:
9     mov   eax, DWORD PTR [rbp-8]
10    add   DWORD PTR [rbp-4], eax
11    add   DWORD PTR [rbp-8], 1
12
13 .L2:
14    cmp   DWORD PTR [rbp-8], 0
15    jns  .L3
16    mov   eax, DWORD PTR [rbp-4]
17    pop   rbp
18    ret

```

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

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.

Shrnutí přednášky

Příklad použití OpenMP - Maticové násobení 2/2

- Příklad násobení matic 1000×1000 s využitím OpenMP na iCore5 (2 jádra s HT $\sim 4 \times$ výpočetní jednotky).
- Násobení matic 5000×5000 (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:
 - 1× vlákno 0.80 s;
 - 2× vlákna 0.39 s;
 - 4× vlákna 0.24 s.

```

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

Diskutovaná témata

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