MPFR

SPACES/CACAO Project

Mardi 25 janvier 2005

Séminaire SEDRE

INRIA Lorraine

Présentation par Patrick Pélissier (Ingénieur associé)

Projet S.P.A.C.E.S.

- SPACES est un projet bilocalisé entre Paris et Nancy.
- L'acronyme SPACES signifie :
  - Systèmes Polynomiaux, Arithmétiques, Calculs Efficaces et Sûrs
  - Solving Problems through Algebraic Computation and Efficient Software
- La partie parisienne s'est scindée : création du projet SALSA.
- Partie nancéienne prépare un nouveau projet, CACAO :
  - Courbes, Algèbres, Calculs, Arithmétiques des Ordinateurs
  - Curves, Algebra, Computer Arithmetic, and so On

  Thèmes : Courbes algébriques, algèbre linéaire et réseaux euclidiens, arithmétique

The Multiple Precision Floating-Point Reliable Library

- Bibliothèque LGPL en C de calcul flottant en précision arbitraire.
- Garantit l'exactitude du résultat retourné.
- Supporte les 4 principaux modes d'arrondis.
- Basée sur la bibliothèque GMP.
- Portable : résultats identiques quelle que soit la machine.

BUT : Montrer qu'une bibliothèque de calcul flottant en précision arbitraire sémantiquement définie est tout aussi efficace.

Calcul flottant

- Les nombres flottants sont de la forme : $x = s * m * \beta^e$
- s est le signe (± 1), β est la base (usuellement 2 ou 10), m est la mantisse (encodée sur p bits ou chiffres), e est l'exposant ($emin \leq e \leq emax$) et p est la précision.
- Représentation non unique, sauf si on impose une convention sur la mantisse (normalisation).
- Exemple : $-1 * 1.1011100 * 2^{17}$ ($p = 8$ et $\beta = 2$)

Problème du calcul flottant

- Le résultat d'un calcul flottant n'est en général pas représentable ; il faut donc arrondir :

$$1.0/3.0 = 0.33333 \text{ ou } 0.33334\,?$$

- Le réponse peut être différente selon la machine sur laquelle on travaille : problème de portabilité du logiciel.
- Importance de l'arrondi (échec du missile Patriot en 1991).

Norme IEEE - 754 (1)

Création d'une norme en 1985 pour standardiser les calculs flottants en base 2 :

- simple précision ('simple' or 'float'),
- simple précision étendue (obsolète),
- double précision ('double') et
- double précision étendue ('extended').

Norme IEEE - 754 (2)

format	taille	précision	*emin*	*emax*	valeur max
simple	32	23+1 bits	-126	+127	$3.403...10^{38}$
double	64	52+1 bits	-1022	+1023	$1.798...10^{308}$
extended	≥ 79	≥ 64	≤ -16382	≥ 16383	$1.190...10^{4932}$

FIG. 1 – Types définis par l'IEEE-754

Norme IEEE - 754 (3)

- Définition de trois nombres spéciaux : NAN, $\pm\infty$ et ± 0.
- Support des flags :
  inexact, invalid, division par 0, overflow, underflow.
- Nombres dénormalisés (autrement $x \neq y \nRightarrow x - y \neq 0$)
- 4 modes d'arrondis :
  vers $-\infty$, vers $+\infty$, vers 0 et au plus proche.
- Opérations spécifiées : $+$, $-$, $*$, / et $\sqrt{\cdot}$
- Pas de spécifications des fonctions élémentaires (exp, log, etc.)

Comment arrondir ?

Soit $x = 1.101e0 + 1.001e2 = 1.10001e2$ qu'il faut arrondir à 4 bits.
Donc $x^- = 1.100e2 \leq x \leq 1.101e2 = x^+$.

- Vers $-\infty$: $\lfloor x \rfloor = 1.100e2$
- Vers $+\infty$: $\lceil x \rceil = 1.101e2$
- Vers 0 : $Z(x) = 1.100e2$
- Au plus proche : $\lfloor x \rceil = 1.100e2$

Pour l'arrondi au plus proche, en cas d'égalité entre $x - x^-$ et $x^+ - x$, on choisit celui dont la mantisse est paire.

$\Longrightarrow$ Le résultat d'une opération est parfaitement spécifié (un seul résultat possible) : arrondi correct.

La norme IEEE-754 en pratique

Un programme n'utilisant que ces 5 opérations est portable sous réserve :
- qu'il n'y ait pas de précision intermédiaire étendue.
- que le langage n'autorise pas de réordonner les opérations.

Le langage C ne permet pas le réordonnancement (sauf contre-ordre donné au compilateur), mais sous x86 il y a une précision intermédiaire étendue.

$\Longrightarrow$ Problème de portabilité du calcul flottant : il faut donner des options de compilation spéciales pour forcer le comportement IEEE-754 strict (GCC : -ffloat-store, ICC : -fp_port -mp, CL : /Op).

Ce problème a disparu sous AMD64 (Utilisation des instructions SSE2 au lieu du FPU).

MPFR : extension en précision arbitraire

format	taille	précision	*emin*	*emax*	valeur max
simple	32	23+1 bits	-126	+127	$3.403...10^{38}$
double	64	52+1 bits	-1022	+1023	$1.798...10^{308}$
extended	≥ 79	≥ 64	≤ -16382	≥ 16383	$1.190...10^{4932}$
MPFR	var	$2 \leq p \leq 2^{31}$	$> -2^{30}$	$< 2^{30}$	$2...10^{323228496}$

FIG. 2 – Types définis par l'IEEE-754 et MPFR (Machine 32 bits)

Mais MPFR ne gère pas les dénormalisés (Utilité ?).

Polynôme de Rump

Le format 'double' ne suffit pas toujours :

$$\frac{1355}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

Pour évaluer le signe de cette expression en $x = 77617$ et en $y = 33096$, il faut un minimum 122 bits (sans réordonner les termes).

MPFR : extension de GMP

GMP est la GNU Multiple Precision Arithmetic Library (bibliothèque C multiprécision la plus rapide disponible sous LGPL). Voir
`http://www.swox.com/gmp/`

Elle est essentiellement découpée en 4 couches :

- MPN : fonctions bas niveaux, souvent écrites en assembleur, travaillant sur des entiers positifs (sans gestion de la mémoire).
- MPZ : 140 fonctions sur les entiers signés.
- MPQ : 35 fonctions sur les rationnels.
- MPF : 64 fonctions sur des flottants en précision arbitraire sans spécification, ni contrôle d'arrondi.

MPFR Convention : Forme générale des fonctions

La forme générale d'une fonction MPFR est la suivante :

```
mpfr_t dest, op1, op2;
inexact = mpfr_mul (dest, op1, op2, rnd_mode);
```

- dest est la destination du calcul.
- op1 et op2 sont les entrées.
- rnd_mode, de type mp_rnd_t, est le mode d'arrondis voulu.
- inexact, de type int, est la valeur ternaire retournée.

La destination et les entrées peuvent être les mêmes variables :

```
mpfr_mul (x, x, x, rnd_mode);
```

MPFR Convention : Les modes d'arrondis

Les modes d'arrondis possibles (type mp_rnd_t) sont :

- GMP_RNDN : Arrondi au plus proche (Nearest).
- GMP_RNDZ : Arrondi vers zéro (Zero).
- GMP_RNDD : Arrondi vers $-\infty$ (Down).
- GMP_RNDU : Arrondi vers $+\infty$ (Up).

MPFR Convention : La valeur ternaire

La valeur ternaire retournée indique la direction de l'erreur faite lors de l'arrondi :

- $inexact = 0$: $dest$ est la valeur exacte $\lfloor x \rceil = x$.
- $inexact > 0$: $dest$ est plus grand que la valeur exacte $\lfloor x \rceil > x$.
- $inexact < 0$: $dest$ est plus petit que la valeur exacte $\lfloor x \rceil < x$.

La valeur exacte est celle obtenue lors du calcul avant l'arrondi, en considérant les entrées comme elles-aussi exactes.

MPFR : Utilisation (1)

```
#include <stdio.h>
#include <gmp.h>    /* Inclusion du header de GMP */
#include <mpfr.h>   /* Inclusion du header de MPFR */
int main () {
  unsigned int i;
  mpfr_t s, t, u;   /* Declaration de variable de type mpfr_t */

  mpfr_init2 (s, 200); /* Initialise s, t, u */
  mpfr_init2 (t, 200); /* avec une mantisse de 200 bits */
  mpfr_init2 (u, 200); /* chaque variable a sa propre precision */
  /* Met s et t a 1 - set unsigned integer */
  mpfr_set_ui (t, 1, GMP_RNDN);
  mpfr_set_ui (s, 1, GMP_RNDN);
```

MPFR : Utilisation (2)

```
  for (i = 1; i <= 100; i++) {
     mpfr_mul_ui (t, t, i, GMP_RNDU); /* multiply unsigned integer */
     mpfr_ui_div (u, 1, t, GMP_RNDD); /* divide unsigned integer */
     mpfr_add (s, s, u, GMP_RNDD);
  }

  printf ("Sum is ");
  mpfr_out_str (stdout, 10, 0, s, GMP_RNDD);
  putchar('\n');
  mpfr_clear(s); /* Libere les variables allouees */
  mpfr_clear(t);
  mpfr_clear(u);
}
```

MPFR : Utilisation (3)

En supposant que MPFR soit installée sur votre machine, on peut compiler ce programme [a] avec :

```
morpork sedre 54 % gcc test.c -lmpfr -lgmp
morpork sedre 55 % ./a.out
Sum is 2.71828182845904523536028747135266249775724709369995957496691 3
```

[a] On peut récupérer le source sur `htpp://www.mpfr.org/sample.html`

Ce que ne fait pas MPFR

Prenons un calcul de produit scalaire : $((x_1 x_2) + (y_1 y_2)) + (z_1 z_2)$

MPFR assure la sémantique de chaque opération :

- pour $\times$: pas de problème.
- pour $+$: les entrées sont des variables intermédiaires approchées : problème de cumul des erreurs.

$\Longrightarrow$ Il faut un système de contrôle des erreurs :

- statique : connaissant l'équation, on précalcule un majorant de l'erreur, et on la câble en dur dans le code (plus rapide).
- dynamique : on utilise une arithmétique d'intervalles pour calculer l'expression et l'erreur commise (plus précis).

Les fonctions mathématiques disponibles

- exp, 2^x, 10^x, log, log2, log10, pow
- cos, sin, tan, arccos, arcsin, arctan
- cosh, sinh, tanh, arccosh, arcsinh, arctanh
- fact, gamma, zeta, erf, agm
- $\log 2$, π et γ

MPFR et GNU

L'INRIA a cédé les droits de MPFR à la Free Software Foundation :

- Utilisation des outils GNU : Autotools, GCC, GCOV, Gprof, etc.
- Respect des GNU coding standards :
  `http://www.gnu.org/prep/standards/`
- Voir aussi Information For Maintainers of GNU Software :
  `http://www.gnu.org/prep/maintain/maintain.html`
- Installation Unix classique :
  ./configure ; make ; make install

Rappel but MPFR

- Comment être fiable ? ($\Leftarrow$ Sémantique précise)
- Comment être portable ?
- Comment être efficace ?

Comment être portable ?

- Respecter au maximum le standard C-89 et éviter les comportements indéfinis.
- Limiter les Warnings (GCC : -Wall -W -ansi -pedantic).
- Éviter d'utiliser les doubles.
- Éviter les hacks.
- Éviter les bugs connus des systèmes (les mots clés $near$ et far sous Visual, $LONG\_MIN/1$ sous FreeBSD 5.2.1/alpha, ...).
- Tester sur le plus de systèmes différents possibles.
- Avoir une Test Suite exhaustive (tout le code doit être testé).

Systèmes supportés par MPFR [a]

- Machines :

  x86 ia64 sparc amd64 alpha s390 hpaa

  powerpc m68k arm

- Systèmes d'exploitation :

  Linux FreeBSD NetBSD win32 HP-UX OSF1 Solaris

- Compilateurs :

  GCC ICC Visual C++ Digital C Compaq C

[a] Liste non exhaustive.

Comment être efficace ?

- Travailler au niveau mot machine, et non pas bit.
- Utiliser la couche MPN de GMP, souvent optimisée en assembleur.
- Implantation de qualité.
- Réduire l'overhead des fonctions.
- Algorithmes efficaces.
- Bonne estimation de la précision intermédiaire des calculs.

Applications typiques utilisant MPFR [a]

- Bibliothèque d'arithmétique d'intervalles multiprécision.
- Vérification des résultats obtenus en double précision (vérifier la stabilité numérique d'un algorithme).
- Calculer les fonctions mathématiques de façon exacte en 53 bits.
- Calculer sur des flottants avec plus de précision.

[a] sans être exhaustif encore une fois

Applications utilisant MPFR [a]

- MPFI : Bibliothèque d'arithmétique d'intervalles multiprécision.
- iRRAM : error-free real arithmetic library.
- The Magma Computational Algebra System
- Computational Geometry Algorithms Library
- Genius Math Tool and the GEL Language
- Giac/Xcas (computer algebra system)
- GNU Fortran 95 (composant GCC 4.0)

[a] parmi celles que l'on connaît

Les concurrents de MPFR

Il n'existe pas de réels concurrents (il manque souvent l'arrondi exact).
Néanmoins, parmi les bibliothèques/logiciels permettant le calcul flottant en précision arbitraire, on peut citer :

- Logiciels commerciaux : Maple, Mathematica, Magma, Mupad.
- Logiciels/bibliothèques sous GPL : PARI/GP, NTL, CLN
- Bibliothèques sous LGPL : GMP/MPF
- Usage non-commercial : Arprec, Lidia
- Freeware : MAPM

Comparaison petite précision (53 bits)

func	MPFR	MPF	PARI	CLN	NTL	ARPREC	MAPM
add	**125**	126	232	398	777	1410	960
sub	**140**	**140**	259	669	850	1590	592
mul	**161**	181	213	450	649	2557	4131
div	**377**	568	452	917	1416	3397	13762
sqrt	**588**	988	3067	1067	5354	21630	98942
cmp	**27**	*na*	34	67	706	246	37
set	**39**	46	47	60	37	145	66

FIG. 3 – Résultats en cycles sur Athlon (distribution logarithmique).

Du côté de la grande précision (10000 digits)

func	mathematica	magma	mpfr	pari	pari-2.2.7	cln
mult	0.8	4.8	**0.52**	3.0	0.53	0.83
div	2.5	5.1	**1.1**	3.9	1.2	2.4
sqrt	1.2	16	**0.79**	13	4.3	1.6
exp	**56**	1000	72	630	130	75
log	51	1600	**35**	1200	500	84
sin	150	1300	**100**	810	160	130
cos	150	1300	**100**	770	150	120
acos	230	3600	760	2700	810	**160**
atan	220	3600	650	2700	800	**160**

FIG. 4 – Résultats en ms sur Athlon MP 2200+.

L'avenir de MPFR

- Des corrections de bugs pour des précisions supérieures à 2^{31} bits.
- Prouver des parties du code de MPFR.
- Optimisation diverses et variées.
- Nouvelles fonctions mathématiques.

Pour conclure, essayez MPFR !

Rendez-vous sur le site :

`http://www.mpfr.org/`

Bonus : Détails techniques (1)

```
typedef struct {
  mpfr_prec_t  _mpfr_prec; /* ~ unsigned long */
  mpfr_sign_t  _mpfr_sign; /* ~ int */
  mp_exp_t     _mpfr_exp;  /* ~ long */
  mp_limb_t   *_mpfr_d;    /* ~ unsigned long* */
} __mpfr_struct;
typedef __mpfr_struct mpfr_t[1];
```

Définir un mpfr_t comme une 'struct [1]' permet d'allouer cette structure lors de la déclaration, mais de passer le pointeur comme paramètre des fonctions (comme le passage par référence sous C++).

Détails techniques (2)

On peut classer les fonctions de MPFR en deux types :

- Fonctions de bas niveau utilisant la couche MPN (mpfr_add, mpfr_sub, etc.)
- Fonctions mathématiques (mpfr_exp, mpfr_log, etc.) utilisant les autres fonctions de MPFR ou des fonctions de MPZ.

Le Dilemme du Fabricant de Tables (1)

Considérons un système en base 2, mantisse de p bits, une fonction élémentaire f (exp, log, etc).
Soit x un nombre et y tels que $|y - f(x)| \leq 2^{EXP(y)-m}$ avec $m \geq p$.

Obtient-on l'arrondi de f(x) en précision p en arrondissant y ?

Le Dilemme du Fabricant de Tables (2)

Pas toujours...

– Arrondi au plus près :

$$y = \overbrace{\underbrace{1.xxx}_{p\ bits}1000}^{m\ bits}xxxx \text{ ou } y = \overbrace{\underbrace{1.xxx}_{p\ bits}0111}^{m\ bits}xxxx$$

– Arrondis dirigés :

$$y = \overbrace{\underbrace{1.xxx}_{p\ bits}0000}^{m\ bits}xxxx \text{ ou } y = \overbrace{\underbrace{1.xxx}_{p\ bits}1111}^{m\ bits}xxxx$$

$\Longrightarrow$ Dilemme du Fabricant de Tables (TMD)

Le Dilemme du Fabricant de Tables (3)

Par exemple, dans Handbook of Mathematical Functions by Abramowitz and Stegun, table 1.1, il y a 4 erreurs d'arrondi [a] :

- $10^{1/3} = 2.154434690031883721\underline{9}$ devrait être 218 (217[592])
- $10^{1/5} = 1.584893192461113485\underline{3}$ devrait être 852 (852[021])
- $100^{1/5} = 2.511886431509580111\underline{2}$ devrait être 111 (110[850])
- $1000^{1/4} = 5.623413251903490804\underline{0}$ devrait être 039 (039[495])

[a] Résultat trouvé par Paul Zimmermann

Stratégie de Ziv

- Traite les cas spéciaux (NAN, $\pm\infty$ et ± 0).
- Traite les autres cas particuliers (cas exacts, ± 1, ...)
- Fixe l'intervalle des exposants au maximum pour éviter les overflows.
- Calcul de la précision initiale pour les calculs intermédiaires
- Initialisation des variables intermédiaires
- **Boucle** ∞
  - Calcul aproximation du résultat et de l'erreur commise
  - Si on peut arrondir, on quitte la boucle (haute probabilité)
  - Mise a jour de la précision intermédiaire
  - Mise a jour de la précision des variables
- $x = arrondi(approximation, rnd\_mode)$
- Libère les variables intermédiaires.
- Restaure la plage des exposants initiale et vérifie les débordements.

Comment on mesure des cycles sur x86 ?

- Utiliser l'instruction *rdtsc* pour faire une mesure précise. Mais il faut savoir l'utiliser :

```
xor eax,eax      ; Fonction 0 de l'instruction cpuid
cpuid            ; Instruction de serialisation : infos du processeur
rdtsc            ; Lecture du compteur de cycle 64 bits dans edx:eax
mov eax,cycle    ; Sauve le compteur de cycle
mov edx,cycle+4  ; dans une variable en memoire
xor eax,eax      ; Refonction 0
cpuid            ; Reappel a une instruction de serialisation
```

- Encadrer le code à mesurer de deux appels à *rdtsc*.
- Enlever l'overhead engendré par la mesure.
- Filtrer pour éliminer les aléas (répéter et garder le minimum).