A Brief Introduction to MPIGMP & MPIBNCpack

Tomonori KOUYA
tkouya@na-net.ornl.gov

Version 0.1: April 10, 2008

1 Introduction

BNCpack is a elementary numerical computation library based on IEEE754 double precision and multiple precision arithmetics with MPFR/GMP. MPIBNCpack is parallelized BNCpack with MPI, which is build on tiny library, MPIGMP, that can transport some multiple presicion variables of MPFR and GMP by packing them into one continous memory area.

Until 2009-04-10, we have confirmed that our MPIBNCpack is executable on following MPI environments:

- (PIII) Intel Pentium III 1GHz + Vine Linux 3.2 + MPICH 1.2.7p1
- (PD) Intel Pentium D 820 + CentOS 4.4/5.2 x86_64 + LAM/MPI
- AMD Athlon64X2 3400+ + Fedora Core 4 x86_64 + LAM/MPI
- (Ci7) Intel Core2 i7 920 + CentOS 5.3 x86_64 + OpenMPI 1.2.7

BNCpack, MPIBNCpack and MPIGMP library are and will be distributed on LGPL v2[4]. WE PRVIDE NO WARRANTY to use them. Bug reports related to our libraries are always welcome. Please send your reports to the e-mail address on the title of this document if you notice anything.

2 Anatomy of MPIBNCpack

2.1 GMP and MPFR

We firstly recommend to read some documents on `http://gmplib.org/` and `http://www.mpfr.org/` in order to know all about GMP and MPFR. You can try to use MPFR functions on "Try MPFR !", `http://ex-cs.sist.ac.jp/~tkouya/try_mpfr.html` if you have not recognized the difference between GMP and MPFR yet.

MPIBNCpack supports to use mpf_t variables of GMP and mpfr_t variables of MPFR. The structures of the two multiple precision floating-point datatypes are shown in Figure 1. These are based on GMP 4.1.2 and MPFR 2.4.1 or later, so these may be changed in future.

The both datatypes have parts of extened sign, exponent and mantissa. The user-defined precision fix the length of mantissa that the pointers _mp_d/_mpfr_d point to.

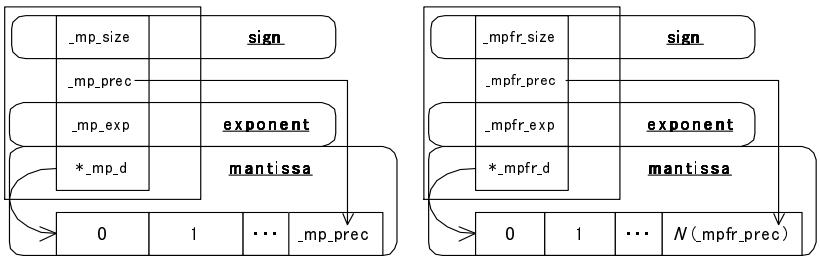

Figure 1: The sturctures of ”mpf_t” datatype in GMP, and ”mpfr_t” datatype in MPFR

Due to these structures, multiple precision variables may not be guranteed to be always stored in continous memory areas.

Therefore, multiple precision data must be packed in MPIBNCpack before transfering among PEs. Figure 2 shows the way of this operation.

For example, we suppose to send one mpfr_t datatype from PE0 to PE1. We pack it (with pack_mpf funciton) into the allocated buffer that void type pointer points to and then unpack it (with unpack_mpf function) into the original variables after PE1 receives. MPIGMP library collects the minimum set of functions necessary for these works, and MPIBNCpack is based on BNCpack for serial computation and MPIGMP for parallel computation. Figure 3 shows the structure of software layers around MPIBNCpack.

2.2 Sample source code with MPIBNCpack

When you want to use MPIBNCpack, especially in order to execute multiple precsion floating-point parallelized computations on MPI cluster, you must prepare MPIGMP Library to do it. In the rest of this section, we show a concrete example code to use MPIGMP functions and MPIBNCpack. Please see such books mainly described about MPI[6], if you are not familiar with MPI functions.

1. You must include the following header files and link the appropriate libraries

```
#include <stdio.h>
#include <math.h> // -lm
#include "mpi.h" // MPI -> libmpi.a and others
#include "bnc.h" // BNCpack -> libbnc.a
#include "mpi_gmp.h" // MPIGMP Library -> libmpfr.a, libgmp.a
#include "mpi_bnc.h" // MPIBNCpack -> libbncmpi.a
```

2. First, initialize MPI functonarity. After that, indicate the communicator (like `MPI_COMM_WORLD`) and then get the numbers of PEs(stored in `numprocs`) and the rank (stored in `myid`) on which is being executed.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
```

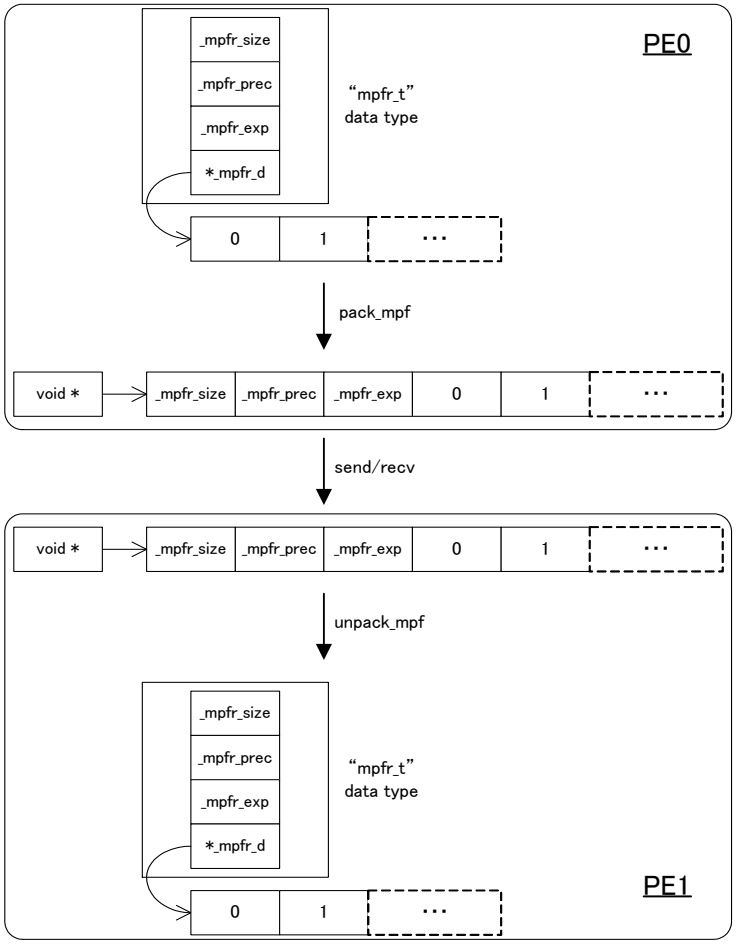

Figure 2: Sending and receiving one mpfr_t variable

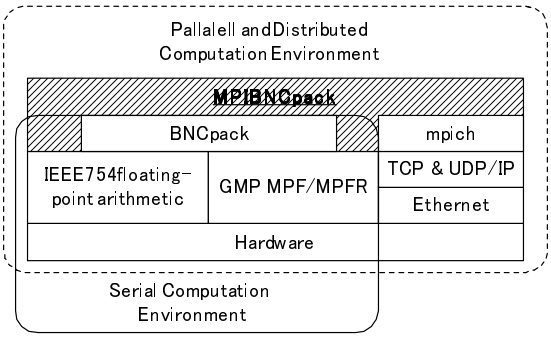

Figure 3: The location of MPIBNCpack in software layers

3. In order to define mutiple precision datatype available in the current communicator, indicate the precision (in binary). In case of this, it is of 256 bits. In addition, define the operation of MPI_Reduce and so on. The current MPIGMP does not support others except summation.

```
#define MPF_PREC 256
    mpf_set_default_prec(MPF_PREC);
    commit_mpf(&(MPI_MPF), MPF_PREC, MPI_COMM_WORLD);
    create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, MPI_COMM_WORLD);
```

4. Describe your calculation running on each process and then get started to communicate in the way as follows:

   (a) Allocate a buffer enough to pack multiple precision variables with allocbuf_mpf function.

   (b) Pack them into the buffer previously allocated.

   (c) Get started to send/receive the packed data in the buffer.

   (d) When completed, unpack them into the original multiple precision variables.

   MPIGMP can just only handle packed data by pack_mpf function. The following list shows that (1) the multiple precision variables, `mpf_mypi` and `mpf_pi` are stored in the buffers, `packed_mpf_mypi` and `packed_mpf_pi`, (2) MPI_Reduce function calls sums up, stores the result in `packed_mpf_pi`, and (3) unpack it into the originarl position, `mpf_pi`.

```
    packed_mpf_mypi = allocbuf_mpf(mpf_get_prec(mpf_mypi), 1);
    packed_mpf_pi = allocbuf_mpf(mpf_get_prec(mpf_pi), 1);
    pack_mpf(mpf_mypi, 1, packed_mpf_mypi);
    pack_mpf(mpf_pi, 1, packed_mpf_pi);
    MPI_Reduce(packed_mpf_mypi, packed_mpf_pi, 1, MPI_MPF, MPI_MPF_SUM, 0, M
PI_COMM_WORLD);
    unpack_mpf(packed_mpf_pi, mpf_pi, 1);
```

5. Finalize MPI, and free the allocated memory areas to store multiple precision datatypes and summation operation previously defined.

```
    free_mpf(&(MPI_MPF));
    free_mpf_op(&(MPI_MPF_SUM));
    MPI_Finalize();
```

3 MPIGMP Library mpi_gmp.h, mpi_gmp.c

See `cpi-gmp.c` if you know how to use these functions.

void *allocbuf_mpz(mpz_t `zdata`**)**

Return the pointer to allocated buffer in which multiple precision integer (`zdata`) will be stored.

void *allocbuf_mpq(mpq_t `qdata`**)**

Return the pointer to allocated buffer in which multiple precision rational number `qdata` will be stored.

void *allocbuf_mpf(unsigned long `prec`**, int** `incount`**)**

Return the pointer to allocated buffer in which `incount` multiple precision floating-point numbers of `prec` bit-length will be stored. Use standard free function if it is released.

long int _mpi_divide_dim(long int `d_dim[]`**, long int** `dim`**, int** `num_procs`**)**

Calculate the number of dimension that will be distributed on each PEs, return it and store each dimension to the array `d_dim[]`, when dividing arrays, vectors and matrices of the total dimension `dim` to `num_procs` PEs. Due to limitation of MPI collective communication, the dimension distributed to each PE is the same number of it, but actual MPI programs use the different numbers stored in `d_dim[]`.

size_t get_bufsize_mpf(mpf_t `fdata`**, int** `incount`**)**

Calculate and return the size of buffer that is able to store `incount` multiple precision variables `fdata`.

size_t get_bufsize_mpz(mpz_t `zdata`**)**

Calculate and return the size of buffer that is able to store multiple precision integer variable `zadata`.

size_t get_bufsize_mpq(mpq_t `qdata`**)**

Calculate and return the size of buffer that is able to store multiple precision rational number `qadata`.

void pack_mpf(mpf_t `a`**, int** `incount`**, void ***`buf`**)**

Pack the `incount` multiple precision variables `a` into the buffer `buf`.

void pack_mpz(mpz_t `a`**, void ***`buf`**)**

Pack the multiple precision integer `a` into the allocated buffer `buf`.

void pack_mpq(mpq_t `a`**, void ***`buf`**)**

Pack the multiple precision rational number `a` into the allocated buffer `buf`.

void unpack_mpf(void *`buf`**, mpf_t** `ret`**, int** `count`**)**

Unpack the data stored in the buffer `buf` into `count` multiple precision floating-point variables starting at `ret`.

void unpack_mpz(void *`buf`**, mpz_t** `ret`**)**

Unpack the data stored in the buffer `buf` into multiple precision integer variable `ret`.

void unpack_mpq(void *`buf`**, mpq_t** `ret`**)**

Unpack the data stored in the buffer `buf` into multiple precision rational number `ret`.

void commit_mpf(MPI_Datatype *mpi_mpf_t, unsigned long prec, MPI_Comm comm)

Define MPI multiple precision floating-point datatype `mpi_mpf_t` available in the communicator `comm`.

void create_mpf_op(MPI_Op *mpi_mpf_op, void (*func)(void *, void *, int *, MPI_Datatype *) , MPI_Comm comm)

Define MPI operation `mpi_mpf_op` for collective communication, available in the communicator `comm`.

void free_mpf(MPI_Datatype *mpi_mpf_t)

Free MPI multiple precision floating-point datatype `mpi_mpf_t`.

void free_mpf_op(MPI_Op *mpi_mpf_op)

Free MPI operation `mpi_mpf_op` for collective communication.

void _mpi_mpf_add(void *in, void *ret, int *len, MPI_Datatype *datatype)

The summation function for MPI collective communication.

4 Complex number arithmetic: mpi_complex.c

The sample program that use these following functions is test_mpidka.c.

MPI_BNC_MPFCMPLX

Predefined macro of multiple precision complex datatype.

void commit_mpi_mpfcmplx(MPI_Datatype *mpfcmplx_t, unsigned long prec, MPI_Comm comm)

Define `prec` bit length multiple precision complex datatype (`mpfcmplx_t`) available in MPI communicator(`comm`).

void free_mpi_mpfcmplx(MPI_Datatype *mpfcmplx_t)

Free multiple precision complex datatype `mpfcmplx_t`.

size_t get_bufsize_mpfcmplx(MPFCmplx a, int incount)

Return the buffer size needed to be stored `incount` multiple precision complex variables (`a`).

void *allocbuf_mpfcmplx(unsigned long prec, int incount)

Allocate `prec` bits length multiple precision complex variables of `incount`, and return the pointer to it.

int pack_cmpfarray(CMPFArray array, void *buf)

Pack multiple precision complex number array `array` into the allocated buffer `buf`.

void unpack_cmpfarray(void *buf, CMPFArray array, long int size)

Unpack the data stored in the buffer `buf` into the multiple precision complex variable `array` of dimension `size`.

void *allocbuf_dcmplx(int `incount`**)**

Allocate a buffer that is able to store IEEE754 double precision complex number of `incount`, and return the pointer to it.

int pack_cdarray(CDArray `array`**, void ***`buf`**)**

Pack IEEE754 double precision complex array into previously allocated buffer (`buf`).

void unpack_cdarray(void *`buf`**, CDArray** `array`**, long int** `size`**)**

Unpack the data stored in `buf` into the IEEE754 double precision complex array `array` of dimension `size`.

5 Basic linear computation: mpi_linear.c

The sample code using the following functions is test_mpilinear.c .

DVector _mpi_init_dvector(long `d_dim[]`**, long int** `dimension`**, MPI_Comm** `comm`**)**

Allocate IEEE754 double precision vector of dimension `dimension` in the communicator `comm`, and allocate the divided vectors of dimension `d_dim[]` on each PE.

void _mpi_free_dvector(DVector `vec`**)**

Free IEEE754 double precision vector `vec` allocated in the communicator `comm`.

void _mpi_divide_dvector(DVector `d_vec`**, long int** `d_dim[]`**, DVector src_vec, MPI_Comm** `comm`**)**

Divide IEEE754 double precison vector `src_vec` on PE0 to `d_vec` of dimension `d_dim[]` on other PEs.

void _mpi_collect_dvector(DVector `src_vec`**, long int** `d_dim[]`**, DVector** `d_vec`**, MPI_Comm** `comm`**)**

Collect the divided IEEE754 double precision vectors `d_vec` to the vector `src_vec` on PE0.

void _mpi_init_dmatrix(DMatrix `ret[]`**, long** `d_dim[]`**, long int** `dimension`**, MPI_Comm** `comm`**)**

Allocate IEEE754 double precision columnwise matrices of `dimension` in the communicator `comm`.

void _mpi_free_dmatrix(DMatrix `mat[]`**, MPI_Comm** `comm`**)**

Free IEEE754 double precision matrix `mat[]` allocated in the communicator `comm`.

void _mpi_divide_dmatrix(DMatrix `d_mat[]`**, long int** `d_dim[]`**, DMatrix** `src_mat`**, MPI_Comm** `comm`**)**

Divide the IEEE754 double precision matrix `src_mat` on PE0 to vectors `d_mat[]` on other PEs.

void _mpi_collect_dmatrix(DMatrix src_mat**, long int** d_dim[]**, DMatrix** d_mat[]**, MPI_Comm** comm**)**

Collect the divided IEEE754 double precision matrices d_mat[] to the matrix src_mat on PE0.

double _mpi_ip_dvector(DVector in_a**, DVector** in_b**, MPI_Comm** comm**)**

Parallely calculate the inner product of IEEE754 double precision vectors in_a and in_b.

void _mpi_mul_dmatrix_dvec(DVector ret**, DMatrix** a[]**, DVector** x**, DVector** x_all**, MPI_Comm** comm**)**

Parallely calculate the multiplication of IEEE754 double precision matrix a and the vector x.

MPFVector _mpi_init_mpfvector(long d_dim[]**, long int** dimension**, MPI_Comm** comm**)**

Allocate multiple precision vectors of dimension dimension in the communicator comm.

void _mpi_free_mpfvector(MPFVector vec**)**

Free the multiple precision vector vec allocated in the communicator comm.

void _mpi_divide_mpfvector(MPFVector d_vec**, long int** d_dim[]**, MPFVector** src_vec**, MPI_Comm** comm**)**

Divide the multiple precision matrix src_mat on PE0 to vectors d_mat[] of dimension d_dim[] on other PEs.

void _mpi_collect_mpfvector(MPFVector src_vec**, long int** d_dim[]**, MPFVector** d_vec**, MPI_Comm** comm**)**

Collect the divided multiple precision vectors d_vec to the vector src_vec on PE0.

void _mpi_init_mpfmatrix(MPFMatrix ret[]**, long** d_dim[]**, long int** dimension**, MPI_Comm** comm**)**

Allocate multiple precision square matrices of dimension dimension in the communicator comm.

void _mpi_free_mpfmatrix(MPFMatrix mat[]**, MPI_Comm** comm**)**

Free the multiple precision floating-point matrices mat[] allocated in the communicator comm.

void _mpi_divide_mpfmatrix(MPFMatrix d_mat[]**, long int** d_dim[]**, MPFMatrix** src_mat**, MPI_Comm** comm**)**

Divide the multiple precision matrix src_mat on PE0 to matrices d_mat[] of dimension d_mat[] on other PEs in communicator comm.

void _mpi_collect_mpfmatrix(MPFMatrix src_mat**, long int** d_dim[]**, MPFMatrix** d_mat[]**, MPI_Comm** comm**)**

Collect the divided multiple precision matrices d_mat[] to the matrix src_mat on PE0.

void _mpi_ip_mpfvector(mpf_t ret**, MPFVector** in_a**, MPFVector** in_b**, MPI_Comm** comm**)**

Parallely calculate the inner product of the multiple precision vectors in_a and in_b.

void _mpi_mul_mpfmatrix_mpfvec(MPFVector ret**, MPFMatrix** a[]**, MPFVector** x**,**

MPFVector x_all**, MPI_Comm** comm**)**

Parallely calculate the multiplication of the multiple precision floating-point matrix a and the vector x.

6 Krylov subspace method: mpi_cg.c

The concrete program used functions as follows is test_mpicg.c.

long int _mpi_DCG(DVector local_answer**, DMatrix** local_a[]**, DVector** local_b**,**

double reps**, double** aeps**, long int** maxtimes**, long int** dim**, MPI_Comm** comm**)**

Parallelized Conjugate-Gradient method of IEEE754 double precision in the communicator comm.

long int _mpi_MPFCG(MPFVector local_answer**, MPFMatrix** local_a[]**,**

MPFVector local_b**, mpf_t** reps**, mpf_t** aeps**, long int** maxtimes**, long int** dim**, MPI_Comm** comm**)**

Parallelized Conjugate-Gradient Method of multiple precision in the communicator comm.

7 Durand-Kerner-Aberth Method: mpi_dka.c

The example using functions as follows is test_mpidka.c.

void _mpi_ddka_init(CDArray local_x_init**, DPoly** func**, MPI_Comm** comm**)**

Calculate Aberth's initial guesses in IEEE754 double precision, and distribute them on each PE in the communicator comm.

void _mpi_ddka(long int ***lasttimes, CDArray** ans**, CDArray** local_ans**, CDArray** x_init**,**

CDArray local_x_init**, DPoly** func**, long int** maxtimes**, double** abs_eps**, double** rel_eps**, MPI_Comm** comm**)**

Parallelized Durand-Kerner method of IEEE754 double precision in the communicator comm.

void _mpi_mpf_dka_init(CMPFArray local_x_init**, MPFPoly** func**, MPI_Comm** comm**)**

Calculate Aberth's initial guesses in multple precision, and distribute them to each PE in the communicator comm.

void _mpi_mpf_dka(long int *lasttimes, CMPFArray ans, CMPFArray local_ans,

CMPFArray x_init, CMPFArray local_x_init, MPFPoly func, long int maxtimes, mpf_t abs_eps, mpf_t rel_eps, MPI_Comm comm)

Parallelized Durand-Kerner method of multiple precision in the communicator comm.

8 Numerical Integrator: mpi_integral.c

These following functions are used in test_mpiint.c.

void _mpi_dtrapezoidal_fs(double *ptr_ret, double x_start, double x_end,

double (*func)(double x), long int num_div, MPI_Comm comm)

Numerically integrate the one variable function (func) over the interval [x_start,x_end] by using trapezoidal rule with parallerism in the communicator comm.

void _mpi_mpf_trapezoidal_fs(mpf_t ret, mpf_t x_start, mpf_t x_end,

void (*func)(mpf_t, mpf_t), long int num_div, MPI_Comm comm, MPI_Datatype mpi_mpf_type)

Numerically integrate the one variable function (func) in multiple precision precision by using trapezoidal rule with parallerism in the communicator comm.

9 Send/Receive/Collective Communication: mpi_bcastbnc.c

These following functions are used in test_mpidka.c.

void _mpi_bcast_dpoly(DPoly poly, MPI_Comm comm)

Broadcast IEEE754 double precision real polynomial poly in the communicator (comm).

void _mpi_bcast_mpfpoly(MPFPoly poly, MPI_Comm comm)

Broadcast multiple precision real polynomial poly in the communicator comm.

10 Benckmark tests

In this section, we show the results of square matrix multiplication on 3 different MPI PC clusters listed in section 1.

We use 128 bits precision $A, B \in \mathbb{R}^{512\times512}$, and then multiply AB. "test_mpimm.c" is the source code of parallelized square matrix multiplication, so read it if you want to know in detail.

The wall clock times of matrix multiplication are shown in Table 1, and the speed-up ratios by parallelization in Figure 4.

Table 1: Wall clock time (second)

# PEs	1	2	4	8	16
PIII	252.7	127.1	58.5	67.0	
PD	117.9	63.4	30.5	12.9	7.3
Ci7	34.7	16.5	7.3	3.6	

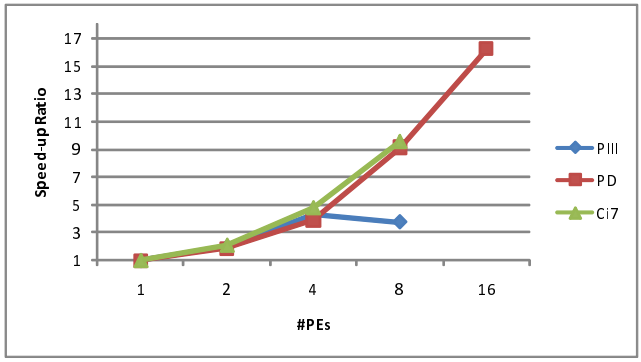

Figure 4: Speed-up ratio by parallelization

Acknowledge

We acknowledge that we have received supports from Shizuoka Insitute of Science and Technology to build BNCpack, MPIBNCpack and MPIGMP from scratch. And also we thank to softwares and hardwares related to our work. Finally, we thank to former professor Hideko Nagasaka for giving her kindness to me.

References

[1] MPIGMP Library, `http://na-inet.jp/na/bnc/mpigmp.tar.gz`

[2] BNCpack(including MPIBNCpack), `http://na-inet.jp/na/bnc/`

[3] GNU MP, `http://swox.com/gmp/`

[4] Lesser GNU Public License, `http://www.gnu.org/copyleft/lesser.html`

[5] MPFR Project, `http://www.mpfr.org/`

[6] P.Pacheco, Parallel Programming with MPI, Morgan Kaufmann Publication, 1996.

[7] Try MPFR!, `http://ex-cs.sist.ac.jp/ tkouya/try_mpfr.html`