第8章　最初のMPIBNCpackプログラミング

本章では，基本線型計算を題材に，1CPU で実行される BNCpack プログラムと，並列実行される MPIBNCpack プログラムの例を示す。計算そのものは単純なものばかりであるが，それをどのように並列分散化するのかを，前章までに扱った 1 対 1 通信，集団通信の例を元に考えて頂きたい。

8.1　数値積分の並列化

5.2 節で用いた台形則を並列化してみる。この場合，(5.1) の関数計算 $f(a)$, $f(b)$, $f(x_i)$ を各 PE で分散して並列に計算することで計算時間の短縮が図れる。最後はこれを和を計算する reduce を用いて集約すれば計算が完了する。

ここでは

$$\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$$

を並列計算するプログラム mpi-int.c と，この多倍長版 mpi-int-gmp.c を以下に示す。

mpi-int.c

```
 1 : #include <stdio.h>
 2 : #include <math.h>
 3 :
 4 : #include "mpi.h"
 5 :
 6 : #include "mpi_bnc.h"
 7 :
 8 : double f(double a)
 9 : {
10 :     return (4.0 / (1.0 + a*a));
11 : }
12 :
13 : int main(int argc,char *argv[])
14 : {
```

```
 15 :     int n, myid, numprocs, i;
 16 :     double pi, start_x, end_x;
 17 :     double startwtime = 0.0, endwtime;
 18 :     int  namelen;
 19 :     char processor_name[MPI_MAX_PROCESSOR_NAME];
 20 :
 21 :     MPI_Init(&argc,&argv);
 22 :     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 23 :     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 24 :
 25 :     n=14 * 16384;
 26 :
 27 :     if(myid == 0) startwtime = MPI_Wtime();
 28 :     start_x = 0.0; end_x = 1.0;
 29 :     _mpi_dtrapezoidal_fs(&pi, start_x, end_x, f, n, MPI_COMM_WORLD);
 30 :     if(myid == 0)
 31 :     {
 32 :         endwtime = MPI_Wtime() - startwtime;
 33 :         printf("BNC: _mpi_dtrapezoidal_fs = %e\n", pi);
 34 :         printf("Time: %f\n", endwtime);
 35 :     }
 36 :
 37 :     MPI_Finalize();
 38 :
 39 :}
```

mpi-int-gmp.c

```
  1 :#include <stdio.h>
  2 :#include <math.h>
  3 :
  4 :#include "mpi.h"
  5 :
  6 :#define USE_GMP
  7 :#define USE_MPFR
  8 :#include "mpi_bnc.h"
  9 :
 10 :void mpf_f(mpf_t ret, mpf_t a)
 11 :{
 12 :     mpf_t tmp;
 13 :
 14 :     mpf_init2(tmp, mpf_get_prec(ret));
 15 :
 16 :     mpf_mul(tmp, a, a);
 17 :     mpf_add_ui(tmp, tmp, 1UL);
 18 :     mpf_ui_div(ret, 4UL, tmp);
```

```
19 :
20 :     mpf_clear(tmp);
21 :
22 :     return;
23 : }
24 :
25 : int main(int argc,char *argv[])
26 : {
27 :     int n, myid, numprocs, i;
28 :     double startwtime = 0.0, endwtime;
29 :     mpf_t mpf_pi, mpf_h, mpf_x;
30 :     int  namelen;
31 :     char processor_name[MPI_MAX_PROCESSOR_NAME];
32 :
33 :     MPI_Init(&argc,&argv);
34 :
35 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
36 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
37 :     create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, MPI_COMM_WORLD);
38 :
39 :     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
40 :     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
41 :
42 :     n=14 * 16384;
43 :
44 :     mpf_init(mpf_h);
45 :     mpf_init(mpf_x);
46 :     mpf_init(mpf_pi);
47 :
48 :     mpf_set_ui(mpf_x, 0UL);
49 :     mpf_set_ui(mpf_h, 1UL);
50 :     if(myid == 0) startwtime = MPI_Wtime();
51 :     _mpi_mpf_trapezoidal_fs(mpf_pi, mpf_x, mpf_h, mpf_f, n, MPI_COMM_WORLD, MPI
52 :     if(myid == 0)
53 :     {
54 :         endwtime = MPI_Wtime() - startwtime;
55 :         printf("BNC: _mpi_mpf_trapezoidal_fs = \n");
56 :         mpf_out_str(stdout, 10, 0, mpf_pi);printf("\n");
57 :         printf("Time: %f\n", endwtime);
58 :     }
59 :
60 :     mpf_clear(mpf_x);
61 :     mpf_clear(mpf_h);
62 :     mpf_clear(mpf_pi);
63 :
64 :     free_mpf(&(MPI_MPF));
```

```
65 :     free_mpf_op(&(MPI_MPF_SUM));
66 :
67 :     MPI_Finalize();
68 :
69 :}
```

8.2　BNCpack プログラムのスケルトン

BNCpack は C(not C++) に準拠したソースプログラムを集積した簡易数値計算ライブラリである。その機能の詳細については公開してあるマニュアル [12] を参照して頂きたい。ここでは基本線型計算を行う，1CPU 用のプログラムの骨組みを簡単に解説する。

BNCpack は実ベクトル (以下，ベクトル)，実正方密行列 (以下，正方行列) のデータ型を持っている。単精度・倍精度・多倍長精度の各データ型は

FVector/FMatrix ･･･ IEEE754 単精度ベクトル型/正方行列型

DVector/DMatrix ･･･ IEEE754 倍精度ベクトル型/正方行列型

MPFVector/MPFMatrix ･･･ 多倍長精度ベクトル型/正方行列型

となっている。本書ではそのうち倍精度・多倍長精度の例のみ示す。

これらベクトル，正方行列を用いた数値計算プログラムは次のような実行手順を取る。

1. ベクトル・行列変数を初期化する。
2. 各成分に数値を代入する。
3. ベクトル・行列を用いた計算を実行する。
4. 変数を出力する。
5. 変数を解放する。

C はガベージコレクションをサポートしていないので，使用した変数の解放処理は必ずしも必要ではないが，やっておくに越したことはない。

以下に示すソースプログラムは全てこのスケルトンに従って構成されている。これを頭に入れた上で読み進めて頂きたい。

8.3 ベクトルの数値計算

まず，10 次元ベクトル $\mathbf{x}$ =[1 2 $\cdots$ 10]^T と $\mathbf{b}$ =[10 9 $\cdots$ 1]^T の和 $\mathbf{c} = \mathbf{x} + \mathbf{b}$ を計算するプログラムの例を示す。最初が倍精度の例である。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "bnc.h"
 6 :
 7 : #define DIM 10
 8 :
 9 : int main()
10 : {
11 :     long int i;
12 :     DVector x, b, c;
13 :
14 :     x = init_dvector(DIM);
15 :     b = init_dvector(DIM);
16 :     c = init_dvector(DIM);
17 :
18 :     for(i = 0; i < DIM; i++)
19 :     {
20 :         set_dvector_i(x, i, (double)(i + 1));
21 :         set_dvector_i(b, i, (double)(DIM - i));
22 :     }
23 :
24 :     add_dvector(c, x, b);
25 :
26 :     print_dvector(c);
27 :
28 :     free_dvector(x);
29 :     free_dvector(b);
30 :     free_dvector(c);
31 :
32 :     return EXIT_SUCCESS;
33 : }
34 :
```

13 行目〜15 行目がベクトル変数の初期化処理にあたる部分で，17 行目〜21 行目が各ベクトルの要素に値を代入している部分，和を計算するのは 23 行目，25 行目が結果を出力している部分で，最後の 27 行目〜29 行目で変数を解放している。

このプログラムをコンパイルする Makefile は次のようになる。

```
1 : CC=gcc
```

```
 2 : DEL=rm
 3 :
 4 : INC=-I/usr/local/include
 5 : LIBDIR=-L/usr/local/lib
 6 : LIB=$(LIBDIR) -lbnc -lmpfr -lgmp -lm
 7 :
 8 : vec1: vec1.c
 9 :     $(CC) -o vec1 vec1.c $(LIB)
10 :
11 : clean:
12 :     -$(DEL) vec1
```

これを実行すると次のような結果を得る。

```
% ./vec1
   0   1.10000000000000000e+01
   1   1.10000000000000000e+01
   2   1.10000000000000000e+01
   3   1.10000000000000000e+01
   4   1.10000000000000000e+01
   5   1.10000000000000000e+01
   6   1.10000000000000000e+01
   7   1.10000000000000000e+01
   8   1.10000000000000000e+01
   9   1.10000000000000000e+01
%
```

次が多倍長精度の例である。ベクトルは全て同じものを使用している。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #define USE_GMP
 6 : #define USE_MPFR
 7 : #include "bnc.h"
 8 :
 9 : #define DIM 10
10 :
11 : int main()
12 : {
13 :     long int i;
14 :     MPFVector x, b, c;
```

```
 15 :
 16 :     set_bnc_default_prec_decimal(50);
 17 :
 18 :     x = init_mpfvector(DIM);
 19 :     b = init_mpfvector(DIM);
 20 :     c = init_mpfvector(DIM);
 21 :
 22 :     for(i = 0; i < DIM; i++)
 23 :     {
 24 :         set_mpfvector_i_d(x, i, (double)(i + 1));
 25 :         set_mpfvector_i_d(b, i, (double)(DIM - i));
 26 :     }
 27 :
 28 :     add_mpfvector(c, x, b);
 29 :
 30 :     print_mpfvector(c);
 31 :
 32 :     free_mpfvector(x);
 33 :     free_mpfvector(b);
 34 :     free_mpfvector(c);
 35 :
 36 :     return EXIT_SUCCESS;
 37 :}
 38 :
```

結果は同じく

```
% ./vec1-gmp
------------------------------------------------------------------------
BNC Default Precision    : 167 bits(50.3 decimal digits)
BNC Default Rounding Mode: Round to Nearest
------------------------------------------------------------------------
    0 1.10000000000000000000000000000000000000000000000000e1
    1 1.10000000000000000000000000000000000000000000000000e1
    2 1.10000000000000000000000000000000000000000000000000e1
    3 1.10000000000000000000000000000000000000000000000000e1
    4 1.10000000000000000000000000000000000000000000000000e1
    5 1.10000000000000000000000000000000000000000000000000e1
    6 1.10000000000000000000000000000000000000000000000000e1
    7 1.10000000000000000000000000000000000000000000000000e1
    8 1.10000000000000000000000000000000000000000000000000e1
    9 1.10000000000000000000000000000000000000000000000000e1
```

となる。
　次に，同じベクトル **x** と **b** を用いて，内積を計算するプログラムの例を示す。構造は全く同じである。

```
 1 :#include <stdio.h>
 2 :#include <stdlib.h>
 3 :#include <math.h>
 4 :
 5 :#include "bnc.h"
 6 :
 7 :#define DIM 10
 8 :
 9 :int main()
10 :{
11 :    long int i;
12 :    double ip;
13 :    DVector x, b;
14 :
15 :    x = init_dvector(DIM);
16 :    b = init_dvector(DIM);
17 :
18 :    for(i = 0; i < DIM; i++)
19 :    {
20 :        set_dvector_i(x, i, (double)(i + 1));
21 :        set_dvector_i(b, i, (double)(DIM - i));
22 :    }
23 :
24 :    ip = ip_dvector(x, b);
25 :
26 :    printf("Inner Produce: %25.17e\n", ip);
27 :
28 :    free_dvector(x);
29 :    free_dvector(b);
30 :
31 :    return EXIT_SUCCESS;
32 :}
33 :
```

　実行結果は

```
% ./vec3
Inner Produce:   2.20000000000000000e+02
%
```

となる。
　多倍長の例は以下の通り。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #define USE_GMP
 6 : #define USE_MPFR
 7 : #include "bnc.h"
 8 :
 9 : #define DIM 10
10 :
11 : int main()
12 : {
13 :     long int i;
14 :     mpf_t ip;
15 :     MPFVector x, b;
16 :
17 :     set_bnc_default_prec_decimal(50);
18 :
19 :     mpf_init(ip);
20 :     x = init_mpfvector(DIM);
21 :     b = init_mpfvector(DIM);
22 :
23 :     for(i = 0; i < DIM; i++)
24 :     {
25 :         set_mpfvector_i_d(x, i, (double)(i + 1));
26 :         set_mpfvector_i_d(b, i, (double)(DIM - i));
27 :     }
28 :
29 :     ip_mpfvector(ip, x, b);
30 :
31 :     printf("Inner Produce: ");
32 :     mpf_out_str(stdout, 10, 0, ip);
33 :     printf("\n");
34 :
35 :     mpf_clear(ip);
36 :     free_mpfvector(x);
37 :     free_mpfvector(b);
38 :
39 :     return EXIT_SUCCESS;
40 : }
41 :
```

実行結果は

```
% ./vec3-gmp
--------------------------------------------------------------------------------
```

```
BNC Default Precision    : 167 bits(50.3 decimal digits)
BNC Default Rounding Mode: Round to Nearest
--------------------------------------------------------------------------------
Inner Produce: 2.200000000000000000000000000000000000000000000000e2
%
```

となる。

8.4　正方行列の数値計算

正方行列を用いた数値計算プログラムも，構造は前述のベクトルを用いたものと全く同じである。ここでは正方行列 X と B を

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}, \quad B = \begin{bmatrix} 25 & 24 & 23 & 22 & 21 \\ 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

とし，その和 $C = X + B$ を計算するプログラムを示す。

```
 1 :#include <stdio.h>
 2 :#include <stdlib.h>
 3 :#include <math.h>
 4 :
 5 :#include "bnc.h"
 6 :
 7 :#define DIM 5
 8 :
 9 :int main()
10 :{
11 :    long int i, j;
12 :    DMatrix c, x, b;
13 :
14 :    x = init_dmatrix(DIM, DIM);
15 :    b = init_dmatrix(DIM, DIM);
16 :    c = init_dmatrix(DIM, DIM);
17 :
18 :    for(i = 0; i < DIM; i++)
19 :    {
20 :        for(j = 0; j < DIM; j++)
21 :        {
```

```
 22 :             set_dmatrix_ij(x, i, j, (double)(i * DIM + j + 1));
 23 :             set_dmatrix_ij(b, i, j, (double)(DIM * DIM - (i * DIM
    + j)));
 24 :         }
 25 :     }
 26 :
 27 :     add_dmatrix(c, x, b);
 28 :
 29 :     print_dmatrix(c);
 30 :
 31 :     free_dmatrix(x);
 32 :     free_dmatrix(b);
 33 :     free_dmatrix(c);
 34 :
 35 :     return EXIT_SUCCESS;
 36 :}
 37 :
```

実行結果は長くなるので省略するが，C の要素は全て 26 となることは容易に分かる。

多倍長化したものは以下の通りである。

```
  1 :#include <stdio.h>
  2 :#include <stdlib.h>
  3 :#include <math.h>
  4 :
  5 :#define USE_GMP
  6 :#define USE_MPFR
  7 :#include "bnc.h"
  8 :
  9 :#define DIM 5
 10 :
 11 :int main()
 12 :{
 13 :    long int i, j;
 14 :    MPFMatrix c, x, b;
 15 :
 16 :    set_bnc_default_prec_decimal(50);
 17 :
 18 :    x = init_mpfmatrix(DIM, DIM);
 19 :    b = init_mpfmatrix(DIM, DIM);
 20 :    c = init_mpfmatrix(DIM, DIM);
 21 :
 22 :    for(i = 0; i < DIM; i++)
 23 :    {
 24 :        for(j = 0; j < DIM; j++)
```

```
25 :         {
26 :             set_mpfmatrix_ij_d(x, i, j, (double)(i * DIM + j + 1)
   );
27 :             set_mpfmatrix_ij_d(b, i, j, (double)(DIM * DIM - (i *
    DIM + j)));
28 :         }
29 :     }
30 :
31 :     add_mpfmatrix(c, x, b);
32 :
33 :     print_mpfmatrix(c);
34 :
35 :     free_mpfmatrix(x);
36 :     free_mpfmatrix(b);
37 :     free_mpfmatrix(c);
38 :
39 :     return EXIT_SUCCESS;
40 :}
41 :
```

次に，同じ行列 X にベクトル $\mathbf{b} = [5\ 4\ 3\ 2\ 1]^T$ を乗じたベクトル $\mathbf{c}$ を計算するプログラムを示す。

```
 1 :#include <stdio.h>
 2 :#include <stdlib.h>
 3 :#include <math.h>
 4 :
 5 :#include "bnc.h"
 6 :
 7 :#define DIM 5
 8 :
 9 :int main()
10 :{
11 :    long int i, j;
12 :    DMatrix x;
13 :    DVector vc, vb;
14 :
15 :    x = init_dmatrix(DIM, DIM);
16 :
17 :    vb = init_dvector(DIM);
18 :    vc = init_dvector(DIM);
19 :
20 :    for(i = 0; i < DIM; i++)
21 :    {
22 :        set_dvector_i(vb, i, (double)(DIM - i));
23 :        for(j = 0; j < DIM; j++)
24 :        {
```

```
 25 :             set_dmatrix_ij(x, i, j, (double)(i * DIM + j + 1));
 26 :         }
 27 :     }
 28 :
 29 :     mul_dmatrix_dvec(vc, x, vb);
 30 :
 31 :     print_dvector(vc);
 32 :
 33 :     free_dmatrix(x);
 34 :
 35 :     free_dvector(vb);
 36 :     free_dvector(vc);
 37 :
 38 :     return EXIT_SUCCESS;
 39 :}
 40 :
```

実行結果は以下の通りである。

```
% ./mat3
    0   3.50000000000000000e+01
    1   1.10000000000000000e+02
    2   1.85000000000000000e+02
    3   2.60000000000000000e+02
    4   3.35000000000000000e+02
%
```

これを多倍長化したものは次のようになる。

```
  1 :#include <stdio.h>
  2 :#include <stdlib.h>
  3 :#include <math.h>
  4 :
  5 :#define USE_GMP
  6 :#define USE_MPFR
  7 :#include "bnc.h"
  8 :
  9 :#define DIM 5
 10 :
 11 :int main()
 12 :{
 13 :    long int i, j;
 14 :    MPFMatrix x;
 15 :    MPFVector vc, vb;
 16 :
```

```
 17 :     set_bnc_default_prec_decimal(50);
 18 :
 19 :     x = init_mpfmatrix(DIM, DIM);
 20 :
 21 :     vb = init_mpfvector(DIM);
 22 :     vc = init_mpfvector(DIM);
 23 :
 24 :     for(i = 0; i < DIM; i++)
 25 :     {
 26 :         set_mpfvector_i_d(vb, i, (double)(DIM - i));
 27 :         for(j = 0; j < DIM; j++)
 28 :         {
 29 :             set_mpfmatrix_ij_d(x, i, j, (double)(i * DIM + j + 1)
    );
 30 :         }
 31 :     }
 32 :
 33 :     mul_mpfmatrix_mpfvec(vc, x, vb);
 34 :
 35 :     print_mpfvector(vc);
 36 :
 37 :     free_mpfmatrix(x);
 38 :
 39 :     free_mpfvector(vb);
 40 :     free_mpfvector(vc);
 41 :
 42 :     return EXIT_SUCCESS;
 43 :}
 44 :
```

実行結果は同様に

```
% ./mat3-gmp
------------------------------------------------------------------------
BNC Default Precision    : 167 bits(50.3 decimal digits)
BNC Default Rounding Mode: Round to Nearest
------------------------------------------------------------------------
    0 3.50000000000000000000000000000000000000000000000e1
    1 1.10000000000000000000000000000000000000000000000e2
    2 1.85000000000000000000000000000000000000000000000e2
    3 2.60000000000000000000000000000000000000000000000e2
    4 3.35000000000000000000000000000000000000000000000e2
%
```

となる。

8.5 MPIBNCpack プログラムのスケルトン

これからは今まで示した 1CPU 用の BNCpack プログラムを，MPIBNCpack を使って並列実行するように書き換えたものを示していく。

基本線型計算を並列実行するための手順は，1CPU のそれと基本的には変わらない。但し，PC Cluster は前述したように MIMD 型であるため，ベクトル，行列を各 PE に分散配置する手順が必要となる。多くの MPI プログラムは膨大なデータを最初から分散して配置するようになっているが，MPIBNCpack ではまず PE0 でユーザがデータを作成し，それを MPIBNCpack の関数を用いて自動的に分散配置するようにしている。こうすることで，ユーザがベクトルや行列の分割方法に頭を悩ます必要がなくなる。反面，膨大なデータを 1CPU で一括処理する部分がプログラム中に入るため，データの配置に時間が食われることになる。

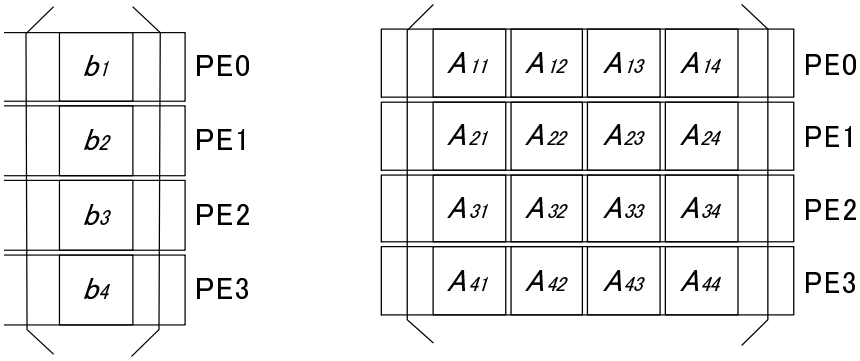

図 8.1: ベクトル・正方行列の分割方法

MPIBNCpack では，ベクトル，正方行列を図 8.1 に示すように，次元数によらず各 PE へ均等に要素を配置する。これは並列処理する基本線型計算の関数内で集団通信を使用しているためである。但し，正方行列は各 PE の中で更に小行列に分割されていることに注意。

従って 5 次元のベクトルであれば，4PE を使って実行すると各 PE へ 2 個ずつ要素がばら撒かれ，合計 8 次元のベクトルに「膨張」してしまう。極端なケースでは，1024PE で実行すると 1024 次元にまで膨張することになる。しかしこれは次元数にジャストフィットする PE 数を考えずに実行した場合に限られる。

MPIBNCpack を用いた基本線型計算を並列実行するプログラムのスケルトンは次のようになる。

1. MPI の初期化処理。

2. PE0 において以下の処理を実行。
   (a) ベクトル・行列変数を初期化する。
   (b) 各成分に数値を代入する。
3. 各 PE において分散配置される変数を初期化する。
4. PE0 の変数を各 PE へ分散配置する。
5. 各 PE においてベクトル・行列を用いた計算を並列実行する。
6. 各 PE の計算結果を PE0 へ集約する。
7. 各 PE において分散配置された変数を解放する。
8. PE0 において以下の処理を実行。
   (a) 変数を出力する。
   (b) 変数を解放する。
9. MPI の終了処理。

8.6　ベクトルの数値計算の並列分散化

ここでは，MPIBNCpack を使ったベクトル計算の並列分散プログラムを示す。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #include "mpi_bnc.h"
 8 :
 9 : #define DIM 10
10 :
11 : int main(int argc, char *argv[])
12 : {
13 :     DVector c, x, b;
14 :     DVector local_c, local_x, local_b;
15 :
16 :     long int d_dim[10];
17 :     long int i, j, local_dim;
```

```
18 :     int myrank, num_procs;
19 :
20 :     MPI_Init(&argc, &argv);
21 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
23 :
24 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
25 :     if(myrank == 0)
26 :     {
27 :         x = init_dvector(local_dim * num_procs);
28 :         b = init_dvector(local_dim * num_procs);
29 :         c = init_dvector(local_dim * num_procs);
30 :
31 :         for(i = 0; i < DIM; i++)
32 :         {
33 :             set_dvector_i(x, i, (double)(i + 1));
34 :             set_dvector_i(b, i, (double)(DIM - i));
35 :         }
36 :     }
37 :     local_x = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
38 :     local_b = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
39 :     local_c = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
40 :
41 :     _mpi_divide_dvector(local_b, d_dim, b, MPI_COMM_WORLD);
42 :     _mpi_divide_dvector(local_x, d_dim, x, MPI_COMM_WORLD);
43 :
44 :     add_dvector(local_c, local_x, local_b);
45 :
46 :     _mpi_collect_dvector(c, d_dim, local_c, MPI_COMM_WORLD);
47 :
48 :     _mpi_free_dvector(local_x);
49 :     _mpi_free_dvector(local_b);
50 :     _mpi_free_dvector(local_c);
51 :
52 :     /* free */
53 :     if(myrank == 0)
54 :     {
55 :         print_dvector(c);
56 :
57 :         free_dvector(x);
58 :         free_dvector(b);
59 :         free_dvector(c);
60 :     }
61 :     MPI_Finalize();
62 :
63 :     return EXIT_SUCCESS;
```

```
64 : }
```

実行結果については，トータルの次元数を除いて前述の vec1.c のそれと同じになるので省略する。
これを多倍長化したものは以下の通りである。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #define USE_GMP
 8 : #define USE_MPFR
 9 : #include "mpi_bnc.h"
10 :
11 : #define DIM 10
12 :
13 : int main(int argc, char *argv[])
14 : {
15 :     MPFVector c, x, b;
16 :     MPFVector local_c, local_x, local_b;
17 :
18 :     long int d_dim[10];
19 :     long int i, j, local_dim;
20 :     int myrank, num_procs;
21 :
22 :     MPI_Init(&argc, &argv);
23 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
25 :
26 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
27 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
28 :
29 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
30 :     if(myrank == 0)
31 :     {
32 :         x = init_mpfvector(local_dim * num_procs);
33 :         b = init_mpfvector(local_dim * num_procs);
34 :         c = init_mpfvector(local_dim * num_procs);
35 :
36 :         for(i = 0; i < DIM; i++)
37 :         {
38 :             set_mpfvector_i_d(x, i, (double)(i + 1));
39 :             set_mpfvector_i_d(b, i, (double)(DIM - i));
40 :         }
```

```
41 :     }
42 :     local_x = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
43 :     local_b = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
44 :     local_c = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
45 :
46 :     _mpi_divide_mpfvector(local_b, d_dim, b, MPI_COMM_WORLD);
47 :     _mpi_divide_mpfvector(local_x, d_dim, x, MPI_COMM_WORLD);
48 :
49 :     add_mpfvector(local_c, local_x, local_b);
50 :
51 :     _mpi_collect_mpfvector(c, d_dim, local_c, MPI_COMM_WORLD);
52 :
53 :     _mpi_free_mpfvector(local_x);
54 :     _mpi_free_mpfvector(local_b);
55 :     _mpi_free_mpfvector(local_c);
56 :
57 :     /* free */
58 :     if(myrank == 0)
59 :     {
60 :         print_mpfvector(c);
61 :
62 :         free_mpfvector(x);
63 :         free_mpfvector(b);
64 :         free_mpfvector(c);
65 :     }
66 :
67 :     free_mpf(&(MPI_MPF));
68 :     MPI_Finalize();
69 :
70 :     return EXIT_SUCCESS;
71 : }
```

これらのプログラムをコンパイルするための Makefile は次のようになる。

```
 1 : CC=mpicc
 2 : DEL=rm
 3 :
 4 : INC=-I/usr/local/include
 5 : LIBDIR=-L/usr/local/lib
 6 : #LIB=$(LIBDIR) -lmpibnc -lmpich -lbnc -lmpfr -lgmp -lm
 7 : LIB=$(LIBDIR) -lmpibnc -lmpi -lbnc -lmpfr -lgmp -lm
 8 :
 9 : all: mpi-vec1 mpi-vec1-gmp
10 :
11 : mpi-vec1: mpi-vec1.c
12 :     $(CC) $(INC) -o mpi-vec1 mpi-vec1.c $(LIB)
```

```
 13 :
 14 : mpi-vec1-gmp: mpi-vec1-gmp.c
 15 :     $(CC) $(INC) -o mpi-vec1-gmp mpi-vec1-gmp.c $(LIB)
 16 :
 17 : clean:
 18 :     -$(DEL) mpi-vec1
 19 :     -$(DEL) mpi-vec1-gmp
```

ベクトルの内積を計算する vec3.c を並列分散化したプログラムは以下の通りである。

```
  1 : #include <stdio.h>
  2 : #include <stdlib.h>
  3 : #include <math.h>
  4 :
  5 : #include "mpi.h"
  6 :
  7 : #include "mpi_bnc.h"
  8 :
  9 : #define DIM 10
 10 :
 11 : int main(int argc, char *argv[])
 12 : {
 13 :     DVector x, b;
 14 :     DVector local_x, local_b;
 15 :
 16 :     long int d_dim[10];
 17 :     double ip;
 18 :     long int i, j, local_dim;
 19 :     int myrank, num_procs;
 20 :
 21 :     MPI_Init(&argc, &argv);
 22 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 23 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
 24 :
 25 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
 26 :     if(myrank == 0)
 27 :     {
 28 :         x = init_dvector(local_dim * num_procs);
 29 :         b = init_dvector(local_dim * num_procs);
 30 :
 31 :         for(i = 0; i < DIM; i++)
 32 :         {
 33 :             set_dvector_i(x, i, (double)(i + 1));
 34 :             set_dvector_i(b, i, (double)(DIM - i));
 35 :         }
 36 :     }
```

```
 37 :     local_x = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
 38 :     local_b = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
 39 :
 40 :     _mpi_divide_dvector(local_b, d_dim, b, MPI_COMM_WORLD);
 41 :     _mpi_divide_dvector(local_x, d_dim, x, MPI_COMM_WORLD);
 42 :
 43 :     ip = _mpi_ip_dvector(local_x, local_b, MPI_COMM_WORLD);
 44 :
 45 :     _mpi_free_dvector(local_x);
 46 :     _mpi_free_dvector(local_b);
 47 :
 48 :     /* free */
 49 :     if(myrank == 0)
 50 :     {
 51 :         printf("Inner Product(MPI): %25.17e\n", ip);
 52 :
 53 :         free_dvector(x);
 54 :         free_dvector(b);
 55 :     }
 56 :     MPI_Finalize();
 57 :
 58 :     return EXIT_SUCCESS;
 59 : }
```

この場合，計算された内積の値は全ての PE で共有されている。
これを多倍長化すると以下のようになる。

```
  1 : #include <stdio.h>
  2 : #include <stdlib.h>
  3 : #include <math.h>
  4 :
  5 : #include "mpi.h"
  6 :
  7 : #define USE_GMP
  8 : #define USE_MPFR
  9 : #include "mpi_bnc.h"
 10 :
 11 : #define DIM 10
 12 :
 13 : int main(int argc, char *argv[])
 14 : {
 15 :     MPFVector x, b;
 16 :     MPFVector local_x, local_b;
 17 :
 18 :     long int d_dim[10];
 19 :     mpf_t ip;
 20 :     long int i, j, local_dim;
```

```
21 :     int myrank, num_procs;
22 :
23 :     MPI_Init(&argc, &argv);
24 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
25 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
26 :
27 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
28 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
29 :     create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, MPI_COMM_WORLD);
30 :
31 :     mpf_init(ip);
32 :
33 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
34 :     if(myrank == 0)
35 :     {
36 :         x = init_mpfvector(local_dim * num_procs);
37 :         b = init_mpfvector(local_dim * num_procs);
38 :
39 :         for(i = 0; i < DIM; i++)
40 :         {
41 :             set_mpfvector_i_d(x, i, (double)(i + 1));
42 :             set_mpfvector_i_d(b, i, (double)(DIM - i));
43 :         }
44 :     }
45 :     local_x = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
46 :     local_b = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
47 :
48 :     _mpi_divide_mpfvector(local_b, d_dim, b, MPI_COMM_WORLD);
49 :     _mpi_divide_mpfvector(local_x, d_dim, x, MPI_COMM_WORLD);
50 :
51 :     _mpi_ip_mpfvector(ip, local_x, local_b, MPI_COMM_WORLD);
52 :
53 :     _mpi_free_mpfvector(local_x);
54 :     _mpi_free_mpfvector(local_b);
55 :
56 :     /* free */
57 :     if(myrank == 0)
58 :     {
59 :         printf("Inner Product(MPI):");
60 :         mpf_out_str(stdout, 10, 0, ip);
61 :         printf("\n");
62 :
63 :         free_mpfvector(x);
64 :         free_mpfvector(b);
65 :     }
66 :
```

```
67 :     free_mpf_op(&(MPI_MPF_SUM));
68 :     free_mpf(&(MPI_MPF));
69 :     MPI_Finalize();
70 :
71 :     return EXIT_SUCCESS;
72 : }
```

8.7 正方行列の数値計算の並列分散化

正方行列の基本線型計算も，ベクトルと同様に並列分散化できる。
例えば，mat1.c を並列分散化したものは以下のようになる。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #include "mpi_bnc.h"
 8 :
 9 : #define DIM 5
10 :
11 : int main(int argc, char *argv[])
12 : {
13 :     long int i, j, local_dim;
14 :     DMatrix c, x, b;
15 :     DMatrix local_c[10], local_x[10], local_b[10];
16 :
17 :     long int d_dim[10];
18 :     int myrank, num_procs;
19 :
20 :     MPI_Init(&argc, &argv);
21 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
23 :
24 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
25 :     if(myrank == 0)
26 :     {
27 :         x = init_dmatrix(num_procs * local_dim, num_procs * local
   _dim);
28 :         b = init_dmatrix(num_procs * local_dim, num_procs * local
   _dim);
29 :         c = init_dmatrix(num_procs * local_dim, num_procs * local
   _dim);
30 :
```

```
31 :         for(i = 0; i < DIM; i++)
32 :         {
33 :             for(j = 0; j < DIM; j++)
34 :             {
35 :                 set_dmatrix_ij(x, i, j, (double)(i * DIM + j + 1)
   );
36 :                 set_dmatrix_ij(b, i, j, (double)(DIM * DIM - (i *
    DIM + j)));
37 :             }
38 :         }
39 :     }
40 :
41 :     _mpi_init_dmatrix(local_c, d_dim, DIM, MPI_COMM_WORLD);
42 :     _mpi_init_dmatrix(local_x, d_dim, DIM, MPI_COMM_WORLD);
43 :     _mpi_init_dmatrix(local_b, d_dim, DIM, MPI_COMM_WORLD);
44 :
45 :     _mpi_divide_dmatrix(local_c, d_dim, c, MPI_COMM_WORLD);
46 :     _mpi_divide_dmatrix(local_x, d_dim, x, MPI_COMM_WORLD);
47 :     _mpi_divide_dmatrix(local_b, d_dim, b, MPI_COMM_WORLD);
48 :
49 :     for(i = 0; i < num_procs; i++)
50 :         add_dmatrix(local_c[i], local_x[i], local_b[i]);
51 :
52 :     _mpi_collect_dmatrix(c, d_dim, local_c, MPI_COMM_WORLD);
53 :
54 :     _mpi_free_dmatrix(local_c, MPI_COMM_WORLD);
55 :     _mpi_free_dmatrix(local_x, MPI_COMM_WORLD);
56 :     _mpi_free_dmatrix(local_b, MPI_COMM_WORLD);
57 :
58 :     if(myrank == 0)
59 :     {
60 :         print_dmatrix(c);
61 :
62 :         free_dmatrix(x);
63 :         free_dmatrix(b);
64 :         free_dmatrix(c);
65 :     }
66 :     MPI_Finalize();
67 :
68 :     return EXIT_SUCCESS;
69 :}
70 :
```

これを多倍長化すると次のようになる。

```
1 :#include <stdio.h>
2 :#include <stdlib.h>
```

```
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #define USE_GMP
 8 : #define USE_MPFR
 9 : #include "mpi_bnc.h"
10 :
11 : #define DIM 5
12 :
13 : int main(int argc, char *argv[])
14 : {
15 :     long int i, j, local_dim;
16 :     MPFMatrix c, x, b;
17 :     MPFMatrix local_c[10], local_x[10], local_b[10];
18 :
19 :     long int d_dim[10];
20 :     int myrank, num_procs;
21 :
22 :     MPI_Init(&argc, &argv);
23 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
25 :
26 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
27 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
28 :
29 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
30 :     if(myrank == 0)
31 :     {
32 :         x = init_mpfmatrix(num_procs * local_dim, num_procs * loc
   al_dim);
33 :         b = init_mpfmatrix(num_procs * local_dim, num_procs * loc
   al_dim);
34 :         c = init_mpfmatrix(num_procs * local_dim, num_procs * loc
   al_dim);
35 :
36 :         for(i = 0; i < DIM; i++)
37 :         {
38 :             for(j = 0; j < DIM; j++)
39 :             {
40 :                 set_mpfmatrix_ij_d(x, i, j, (double)(i * DIM + j
   + 1));
41 :                 set_mpfmatrix_ij_d(b, i, j, (double)(DIM * DIM -
   (i * DIM + j)));
42 :             }
43 :         }
```

```
44 :     }
45 :
46 :     _mpi_init_mpfmatrix(local_c, d_dim, DIM, MPI_COMM_WORLD);
47 :     _mpi_init_mpfmatrix(local_x, d_dim, DIM, MPI_COMM_WORLD);
48 :     _mpi_init_mpfmatrix(local_b, d_dim, DIM, MPI_COMM_WORLD);
49 :
50 :     _mpi_divide_mpfmatrix(local_c, d_dim, c, MPI_COMM_WORLD);
51 :     _mpi_divide_mpfmatrix(local_x, d_dim, x, MPI_COMM_WORLD);
52 :     _mpi_divide_mpfmatrix(local_b, d_dim, b, MPI_COMM_WORLD);
53 :
54 :     for(i = 0; i < num_procs; i++)
55 :         add_mpfmatrix(local_c[i], local_x[i], local_b[i]);
56 :
57 :     _mpi_collect_mpfmatrix(c, d_dim, local_c, MPI_COMM_WORLD);
58 :
59 :     _mpi_free_mpfmatrix(local_c, MPI_COMM_WORLD);
60 :     _mpi_free_mpfmatrix(local_x, MPI_COMM_WORLD);
61 :     _mpi_free_mpfmatrix(local_b, MPI_COMM_WORLD);
62 :
63 :     if(myrank == 0)
64 :     {
65 :         print_mpfmatrix(c);
66 :
67 :         free_mpfmatrix(x);
68 :         free_mpfmatrix(b);
69 :         free_mpfmatrix(c);
70 :     }
71 :
72 :     free_mpf(&(MPI_MPF));
73 :     MPI_Finalize();
74 :
75 :     return EXIT_SUCCESS;
76 : }
77 :
```

行列とベクトルの積を計算する mat3.c も次のように並列分散化できる。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #include "mpi_bnc.h"
 8 :
 9 : #define DIM 5
10 :
```

```
11 : int main(int argc, char *argv[])
12 : {
13 :     long int i, j, local_dim;
14 :     DMatrix x;
15 :     DMatrix local_x[10];
16 :     DVector vc, vb;
17 :     DVector local_vc, local_vb, local_vtmp;
18 :
19 :     long int d_dim[10];
20 :     int myrank, num_procs;
21 :
22 :     MPI_Init(&argc, &argv);
23 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
24 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
25 :
26 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
27 :     if(myrank == 0)
28 :     {
29 :         x = init_dmatrix(num_procs * local_dim, num_procs * local
   _dim);
30 :
31 :         vb = init_dvector(num_procs * local_dim);
32 :         vc = init_dvector(num_procs * local_dim);
33 :         for(i = 0; i < DIM; i++)
34 :         {
35 :             set_dvector_i(vb, i, (double)(DIM - i));
36 :             for(j = 0; j < DIM; j++)
37 :             {
38 :                 set_dmatrix_ij(x, i, j, (double)(i * DIM + j + 1)
   );
39 :             }
40 :         }
41 :     }
42 :
43 :     local_vtmp = init_dvector(num_procs * local_dim);
44 :
45 :     _mpi_init_dmatrix(local_x, d_dim, DIM, MPI_COMM_WORLD);
46 :
47 :     local_vb = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
48 :     local_vc = _mpi_init_dvector(d_dim, DIM, MPI_COMM_WORLD);
49 :
50 :     _mpi_divide_dmatrix(local_x, d_dim, x, MPI_COMM_WORLD);
51 :
52 :     _mpi_divide_dvector(local_vc, d_dim, vc, MPI_COMM_WORLD);
53 :     _mpi_divide_dvector(local_vb, d_dim, vb, MPI_COMM_WORLD);
54 :
```

```
55 :     _mpi_mul_dmatrix_dvec(local_vc, local_x, local_vb, local_vtmp
   , MPI_COMM_WORLD);
56 :
57 :     _mpi_collect_dvector(vc, d_dim, local_vc, MPI_COMM_WORLD);
58 :
59 :     _mpi_free_dmatrix(local_x, MPI_COMM_WORLD);
60 :
61 :     _mpi_free_dvector(local_vc);
62 :     _mpi_free_dvector(local_vb);
63 :
64 :     free_dvector(local_vtmp);
65 :
66 :     if(myrank == 0)
67 :     {
68 :         print_dvector(vc);
69 :
70 :         free_dmatrix(x);
71 :         free_dvector(vb);
72 :         free_dvector(vc);
73 :     }
74 :
75 :     MPI_Finalize();
76 :
77 :     return EXIT_SUCCESS;
78 : }
79 :
```

多倍長化したものは以下の通り。

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>
 3 : #include <math.h>
 4 :
 5 : #include "mpi.h"
 6 :
 7 : #define USE_GMP
 8 : #define USE_MPFR
 9 : #include "mpi_bnc.h"
10 :
11 : #define DIM 5
12 :
13 : int main(int argc, char *argv[])
14 : {
15 :     long int i, j, local_dim;
16 :     MPFMatrix x;
17 :     MPFMatrix local_x[10];
18 :     MPFVector vc, vb;
```

```
19 :     MPFVector local_vc, local_vb, local_vtmp;
20 :
21 :     long int d_dim[10];
22 :     int myrank, num_procs;
23 :
24 :     MPI_Init(&argc, &argv);
25 :     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
26 :     MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
27 :
28 :     _mpi_set_bnc_default_prec_decimal(50, MPI_COMM_WORLD);
29 :     commit_mpf(&(MPI_MPF), ceil(50/log10(2.0)), MPI_COMM_WORLD);
30 :     create_mpf_op(&(MPI_MPF_SUM), _mpi_mpf_add, MPI_COMM_WORLD);
31 :
32 :     local_dim = _mpi_divide_dim(d_dim, DIM, num_procs);
33 :     if(myrank == 0)
34 :     {
35 :         x = init_mpfmatrix(num_procs * local_dim, num_procs * loc
   al_dim);
36 :
37 :         vb = init_mpfvector(num_procs * local_dim);
38 :         vc = init_mpfvector(num_procs * local_dim);
39 :         for(i = 0; i < DIM; i++)
40 :         {
41 :             set_mpfvector_i_d(vb, i, (double)(DIM - i));
42 :             for(j = 0; j < DIM; j++)
43 :             {
44 :                 set_mpfmatrix_ij_d(x, i, j, (double)(i * DIM + j
   + 1));
45 :             }
46 :         }
47 :     }
48 :
49 :     local_vtmp = init_mpfvector(num_procs * local_dim);
50 :
51 :     _mpi_init_mpfmatrix(local_x, d_dim, DIM, MPI_COMM_WORLD);
52 :
53 :     local_vb = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
54 :     local_vc = _mpi_init_mpfvector(d_dim, DIM, MPI_COMM_WORLD);
55 :
56 :     _mpi_divide_mpfmatrix(local_x, d_dim, x, MPI_COMM_WORLD);
57 :
58 :     _mpi_divide_mpfvector(local_vc, d_dim, vc, MPI_COMM_WORLD);
59 :     _mpi_divide_mpfvector(local_vb, d_dim, vb, MPI_COMM_WORLD);
60 :
61 :     _mpi_mul_mpfmatrix_mpfvec(local_vc, local_x, local_vb, local_
   vtmp, MPI_COMM_WORLD);
```

```
62 :
63 :     _mpi_collect_mpfvector(vc, d_dim, local_vc, MPI_COMM_WORLD);
64 :
65 :     _mpi_free_mpfmatrix(local_x, MPI_COMM_WORLD);
66 :
67 :     _mpi_free_mpfvector(local_vc);
68 :     _mpi_free_mpfvector(local_vb);
69 :
70 :     free_mpfvector(local_vtmp);
71 :
72 :     if(myrank == 0)
73 :     {
74 :         print_mpfvector(vc);
75 :
76 :         free_mpfmatrix(x);
77 :         free_mpfvector(vb);
78 :         free_mpfvector(vc);
79 :     }
80 :
81 :     free_mpf_op(&(MPI_MPF_SUM));
82 :     free_mpf(&(MPI_MPF));
83 :
84 :     MPI_Finalize();
85 :
86 :     return EXIT_SUCCESS;
87 :}
88 :
```

演習問題

1. mat3.c を改良し，

$$
\begin{aligned}
A &= [1/(i+j-1)]\ (i, j = 1, 2, ..., 10) \\
\mathbf{b} &= [i]\ (i = 1, 2, ..., 10)
\end{aligned}
$$

という行列 A とベクトル $\mathbf{b}$ を使って

$$\sum_{i=1}^{10} A(i\mathbf{b})$$

を計算するプログラムを作れ。

2. mpi-mat3.c を参考に，上記の計算を並列分散化して実行するプログラムを作れ。