Source Code for the MM optimization algorithm

This section consists of the following sub-sections:

- Description of the algorithm,
- Files and functions used in the application,
- Steps to run the application.

1. Description of the Algorithm

The parameters to the *nettype* definition **ParallelAxB** are number of processors along the row and along the column making a total of $p \times p$ processors, size of block of each matrix r, problem size in $r \times r$ blocks, generalised block size in $r \times r$ blocks, the common heights of the rectangular areas of the generalised block between pairs of processors in $r \times r$ blocks.

/* Description of the algorithm to be filled here */

2. Files and functions used in the application

Shown below are the files and the functions used in the application:

2.1 The mxm.h header file

This file contains a constant, which is used to initialize the matrix elements and a macro to get the common height of the rectangular areas between a pair of processors in the generalized block

```
#ifndef  _MXM_H
#define  _MXM_H

#define  MXM_CONSTANT_NUMBER  2.00
#define  H(a, b, c, d, p)     h[(a*p*p*p+b*p*p+c*p+d)]

#endif
```

2.2 The ParallelAxB.mpc nettype declaration file

Contains the **nettype** declaration of ParallelAxB.

```
#include "mxm.h"

typedef struct {int I; int J;} Processor;
nettype ParallelAxB(int p, int r, int n, int l, int w[p],
                    int h[p*p*p*p])
{
   coord I=p, J=p;
   node {I>=0 && J>=0: bench*
        (w[J]*H(I, J, I, J, p)*(n/l)*(n/l)*n);};
```

```
    link (K=p, L=p)
    {
         I>=0 && J>=0 && I!=K : length*
             (w[I]*H(I, J, I, J, p)*(n/l)*(n/l)*(r*r)*sizeof(double))
            [I, J] -> [K, J];
         I>=0 && J>=0 && J!=L && (H(I, J, K, L, p)>0) : length*
            (w[J]*H(I, J, K, L, p)*(n/l)*(n/l)*(r*r)*sizeof(double))
            [I, J]->[K, L];
    };
    parent[0];
    scheme
    {
       int k;
       Processor Root, Receiver, Current;
       for(k = 0; k < n; k++)
       {
          int Acolumn = k%l, Arow;
          int Brow = k%l, Bcolumn;
          par(Arow = 0; Arow <l; )
          {
             GetProcessor(Arow, Acolumn, p, h, w, &Root);
             par(Receiver.I = 0; Receiver.I < p; Receiver.I++)
                par(Receiver.J = 0; Receiver.J < p; Receiver.J++)
                if((Root.I != Receiver.I || Root.J != Receiver.J) &&
                    Root.J != Receiver.J)
                   if(H((Root.I), (Root.J), (Receiver.I),
                        (Receiver.J), p) > 0)
                     (100/(w[Root.J]*(n/l)))%%
                     [(Root.I), (Root.J)] ->
                     [(Receiver.I), (Receiver.J)];
             Arow += H((Root.I), (Root.J), (Root.I), (Root.J), p);
          }
          par(Bcolumn = 0; Bcolumn < l; )
          {
             GetProcessor(Brow, Bcolumn, p, h, w, &Root);
             par(Receiver.I = 0; Receiver.I < p; Receiver.I++)
                if(Root.I != Receiver.I)
                  (100/
                  (H((Root.I), (Root.J), (Root.I), (Root.J),
                   p)*(n/l))) %%
                  [(Root.I), (Root.J)] -> [(Receiver.I), (Root.J)];
             Bcolumn += w[Root.J];
          }
          par(Current.I = 0; Current.I < p; Current.I++)
              par(Current.J = 0; Current.J < p; Current.J++)
                (100/n) %% [(Current.I), (Current.J)];
       }
    };
};
```

2.3 The mxm.mpc main file

This is the file that contains the main. The essential steps in the execution of the MM application are:

- Getting the inputs to the MM application. These inputs are the problem size in $r \times r$ blocks, the size of the block r and the number of processors along a row or column p,
- Estimation of the actual speeds of the processors using **recon** statement. The function used is *rMxM*,
- Fetching the actual speeds of the processors using the function *MPC_Get_processors_info*,
- Determining the optimal generalized block size using **timeof** operator,
- Broadcasting the optimal generalized block size from host to all the other participating processors,
- Determination of the parameters to the nettype declaration ParallelAxB using the function *DistributeLoad*,
- Creation of a network *g* of network type ParallelAxB,
- Initialization of the matrix elements by the member processes of the network respectively,
- Execution of the parallel algorithm using the function *ExecuteAlgorithm* by the member processes of the network,
- Printing the elapsed parallel execution time on the console associated with the host terminal.

```
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <mpc.h>
#include "ParallelAxB.mpc"
#include "Load_balance.mpc"
#include "mxm_i.mpc"

int [*] main(int [host] argc, char** [host] argv)
{
     repl int p, r, n, l, *w, *h, *trow, *tcolumn, nump;
     repl double* speeds;
     int [host] opt_l;
     double [host] time_i, Elapsed_time;
     double barrier_time, i_barrier_time;

     p = [host]atoi(argv[1]);
     r = [host]atoi(argv[2]);
     n = [host]atoi(argv[3]);

     /*
      * Update the estimation of speeds of the processors
      */
     {
        repl int i, j;
        repl double x[r*r], y[r*r], z[r*r];
        for (i = 0; i < r; i++) {
            for (j = 0; j < r; j++) {
                x[i*r+j] = MXM_CONSTANT_NUMBER;
                y[i*r+j] = MXM_CONSTANT_NUMBER;
```

```
            z[i*r+j] = 0.0;
         }
      }
      recon rMxM(x, y, z, r);
}

/*
 * Detect the total number of physical processors,
 * Detect the speed of the physical processors
 */
nump = MPC_Get_number_of_processors();
speeds = (double*)malloc(sizeof(double)*nump);
MPC_Get_processors_info(NULL, speeds);

/*
 * w --- widths of the rectangular areas of the processes in the
 *       generalized block.
 * h --- common heights of the rectangular areas of the
 *       processes in the generalized block.
 * trow --- Top rows of the rectangular areas of the processes
 *          in the generalized block.
 * tcolumn --- Top columns of the rectangular areas of the
 *             processes in the generalized block.
 */
w = (int*)malloc(sizeof(int)*p);
h = (int*)malloc(sizeof(int)*(p*p*p*p));
trow = (int*)malloc(sizeof(int)*p*p);
tcolumn = (int*)malloc(sizeof(int)*p);

/*
 * Determine the optimal value of the generalized block size.
 */
[host]:
{
   int i;
   double algo_time, min_algo_time = DBL_MAX;
   for (i = p; i < n; i++) {
      DistributeLoad(nump, speeds, p, i, w, h, trow, tcolumn);
      algo_time = timeof(
                  net ParallelAxB(p, r, n, i, w, h) paxb);
      if (algo_time < min_algo_time) {
         opt_l = i;
         min_algo_time = algo_time;
      }
   }
}
/*
 * Start timing the algorithm.
 */
time_i = [host]MPC_Wtime();

/*
 * Broadcast the optimal generalised block size to all the
 * partcipating processors.
 */
l = opt_l;
```

```
/*
 * Determine the widths, common heights, top rows, top columns
 * of the rectangular areas of the processors.
 */
DistributeLoad(nump, speeds, p, l, w, h, trow, tcolumn);
free(speeds);

{
   /*
    * Create the network.
    */
   net ParallelAxB(p, r, n, l, w, h) g;
   int [g] icoord, [g] jcoord;

   /*
    * Initialize the matrix elements and execute the algorithm
    */
   [g]:
   {
      int  x, y, i, j, s, t, MyTopRow, MyTopColumn;
      double *a, *b, *c;

      a = (double*)malloc(sizeof(double)*(n*r)*(n*r));
      b = (double*)malloc(sizeof(double)*(n*r)*(n*r));
      c = (double*)malloc(sizeof(double)*(n*r)*(n*r));

      icoord = I coordof icoord;
      jcoord = J coordof jcoord;

      MyTopRow = trow[icoord*p + jcoord];
      MyTopColumn = tcolumn[jcoord];

      for (x = MyTopRow; x < n; x+=l) {
          for (y = MyTopColumn; y < n; y+=l) {
              for (i = 0;
                   i < H(icoord, jcoord, icoord, jcoord, p);
                   i++) {
                  for (j = 0; j < w[jcoord]; j++) {
                      for (s = 0; s < r; s++) {
                          for (t = 0; t < r; t++) {
                              a[(x*r*n*r) + y*r + (i*r*n*r) +
                                j*r + (s*r*n) + t] =
                               MXM_CONSTANT_NUMBER;
                              b[(x*r*n*r) + y*r + (i*r*n*r) +
                                j*r + (s*r*n) + t] =
                                MXM_CONSTANT_NUMBER;
                              c[(x*r*n*r) + y*r + (i*r*n*r) +
                                j*r + (s*r*n) + t] = 0.0;
                          }
                      }
                  }
              }
          }
      }

      ([(p, r, n, l, w, h) g])ExecuteAlgorithm(
                   icoord, jcoord, a, b, c, trow, tcolumn);
```

```
            {
               free(w);
               free(h);
               free(trow);
               free(tcolumn);
               free(a);
               free(b);
               free(c);
            }

            i_barrier_time = MPC_Wtime();
            ([(0)g])MPC_Barrier();
            barrier_time = MPC_Wtime() - i_barrier_time;
         }
      }

      /*
       * Print the algorithm execution time.
       */
      [host]:
      {
         Elapsed_time = MPC_Wtime() - time_i - barrier_time;
         printf(
            "Problem size=%d, time(sec)=%0.6f, time(min)=%0.6f\n",
            (n*r),
            Elapsed_time,
            (double)Elapsed_time/(double)(60.00)
         );
      }
}
```

2.4 The mxm_i.mpc algorithm implementation file

This file consists of the following functions:

```
/*
 * This function is used in recon to update the estimation of speeds
 * of the processors.
 */
int rMxM (double *a, double *b, double *c, int r) {
    int l, m, t;
    for (l = 0; l < r; l++) {
        for (m = 0; m < r; m++) {
            for (t = 0; t < r; t++) {
                c[l*r + m] += a[l*r + t] * b[t*r + m] ;
            }
        }
    }
    return 0;
}

/*
 * This function gives the common height between a pair of processors
 * in a generalized block.
 */
```

```
int CommonHeight(int top_row_1, int bottom_row_1, int top_row_2,
                 int bottom_row_2) {
  /*
   * One area contains the other
   */
  if ((top_row_1 >= top_row_2) && (bottom_row_1 <= bottom_row_2))
     return (bottom_row_1 - top_row_1);
  if ((top_row_1 <= top_row_2) && (bottom_row_1 >= bottom_row_2))
     return (bottom_row_2 - top_row_2);
  /*
   * One area is followed or preceded by another
   * with an overlap
   */
  if ((top_row_1 <= top_row_2) && (bottom_row_1 >= top_row_2)
      && (bottom_row_1 <= bottom_row_2))
     return (bottom_row_1 - top_row_2);
  if ((top_row_1 >= top_row_2) && (top_row_1 <= bottom_row_2)
      && (bottom_row_1 >= bottom_row_2))
     return (bottom_row_2 - top_row_1);

  return 0;
}

/*
 * This function returns the coordinates of the root processor in the
 * processor grid. The inputs to this function are the row and column
 * of the root processor, the number of processors, the widths and
 * the common heights of the rectangular areas in the generalized
 * block.
 */
int GetProcessor(const int row, const int column, const int p, const
                 int *h, const int *w, Processor* proc) {
    int x, y, i, j, tempy;
    int *trow = (int*)malloc(sizeof(int)*p);
    for (i = 0; i < p; i++) trow[i] = 0;
    for (x = 0; x < p; x++) {
        tempy = 0;
        for (y = 0; y < p; y++)
        {
            int hi = H(x, y, x, y, p);
            int wi = w[y];
            if (x) trow[y] += H((x-1), y, (x-1), y, p);
            for (i = 0; i < hi; i++) {
                for (j = 0; j < wi; j++) {
                    if (((row >= (trow[y] + i)) &&
                         (row < (trow[y] + hi)))&&
                         ((column >= (tempy + j)) &&
                         (column < (tempy + wi)))) {
                       proc->I = x;
                       proc->J = y;
                       free(trow);
                       return 0;
                    }
                }
            }
            tempy += w[y];
        }
```

```
     }
     free(trow);
     return 0;
}

/*
 * Execution of the parallel algorithm
 */
void [net ParallelAxB(p, r, n, l, w, h) g]
ExecuteAlgorithm(int icoord, int jcoord, double *a,
                 double *b, double *c, int *trow, int* tcolumn) {
    int MyTopRow, MyTopColumn;
    int m, s, t, x, y, z;
    repl int i, j, k;
    repl Processor Me, Root, Receiver;

    Me.I = icoord;
    Me.J = jcoord;
    MyTopRow = trow[(Me.I)*p + Me.J];
    MyTopColumn = tcolumn[Me.J];

    for (k = 0; k < n; k++) {
       /*
        * P(i,k) broadcasts a(i,k) to p(i,*) horizontally.
        */
       int Acolumn = (k%l);
       int Brow = (k%l);
       for (i = 0; i < n; i++) {
          int Arow = (i%l);
          GetProcessor(Arow, Acolumn, p, h, w, &Root);
          {
             flex subnet[g:(CommonHeight(Arow, Arow+1,
             trow[I*p+J], (trow[I*p+J] + H(I, J, I, J,p))) > 0)]
             rSubnet;

             [rSubnet]:
             {
                int am_I_root = 0;
                double temp[r*r];
                if (((Root.I) == (Me.I)) && ((Root.J) == (Me.J))) {
                   am_I_root = 1;
                   for (x = 0; x < r; x++)
                      for (y = 0; y < r; y++)
                         temp[x*r + y] = a[i*r*n*r + k*r + x*n*r + y];
                }

                temp[] = [rSubnet:(I==(Root.I) && J==(Root.J))]temp[];

                if (!am_I_root)
                   for (x = 0; x < r; x++)
                      for (y = 0; y < r; y++)
                         a[i*r*n*r + k*r + x*n*r + y] = temp[x*r + y];
             }
          }
       }
       /*
        * P(k,j) broadcasts a(k,j) to p(*,j) vertically.
```

```
         */
        for (j = 0; j < n; j++) {
          int Bcolumn = (j%l);
          GetProcessor(Brow, Bcolumn, p, h, w, &Root);
          {
             flex subnet[g:(J==(Root.J))] cSubnet;
             [cSubnet]:
             {
               int am_I_root = 0;
               double temp[r*r];

               if ((Me.I) == (Root.I)) {
                  am_I_root = 1;
                  for (x = 0; x < r; x++)
                     for (y = 0; y < r; y++)
                        temp[x*r + y] = b[k*r*n*r + j*r + x*n*r + y];
               }

               temp[] = [cSubnet:(I==(Root.I) && J==(Root.J))] temp[];

               if (!am_I_root)
                  for (x = 0; x < r; x++)
                     for (y = 0; y < r; y++)
                        b[k*r*n*r + j*r + x*n*r + y] = temp[x*r + y];
             }
          }
        }
        for (x = MyTopRow; x < n; x+=l) {
           for (y = MyTopColumn; y < n; y+=l) {
              for (i = 0; i < H((Me.I), (Me.J), (Me.I), (Me.J), p);
                   i++) {
                 for (j = 0; j < w[(Me.J)]; j++) {
                    for (s = 0; s < r; s++) {
                       for (m = 0; m < r; m++) {
                          for (t = 0; t < r; t++) {
                           c[x*r*n*r + y*r + i*r*n*r + j*r + s*n*r + m]
                           +=
                           a[x*r*n*r + 0   + i*r*n*r + k*r + s*n*r + t]
                           *
                           b[0       + y*r + k*r*n*r + j*r + t*n*r + m]
                           ;
                          }
                       }
                    }
                 }
              }
           }
        }

        return;
     }
```

2.5 The Load_balance.mpc **load balancing file**

This file contains the function '*DistributeLoad*'. The inputs to this function are the number of processors, actual speeds of the processors, the number of processors along a row and column each and the generalized block size. The outputs from this function are the widths of the rectangular areas of the processors in the generalized block, the common heights of the rectangular areas between a pair of processors in the generalized block, the top rows of the rectangular areas of the processors in the generalized block and the top columns of the rectangular areas of the processors in the generalized block.

```
int DistributeLoad(int nump, double *speeds, int p, int l, int *w,
                   int *h, int *trow, int* tcolumn) {
   int i, j, k, x, y, csum[p], tsum = 0;
   double total = 0.0;
   for (i = 0; i < p; i++) {
       csum[i] = 0;
       for (j = 0; j < p; j++) csum[i] += speeds[j*p + i];
       tsum += csum[i];
   }
   for (i = 0; i < p; i++) {
       for (j = 0; j < p; j++) {
           double temp = ((double)
                          speeds[i*p + j]/(double)csum[j])*l;
           if (temp < 1.0)
               temp = 1.0;
           temp = floor(temp);
           H(i, j, i, j, p) = temp;
       }
   }
   for (j = 0; j < p; j++) {
       total = 0.0;
       for (i = 0; i < p; i++)
           total += H(i, j, i, j, p);
       if (total > l) {
          int ind = 0;
          for (i = 0; i < p; i++) {
              if (H(i, j, i, j, p) > 1.0) {
                  ind++;
                  H(i, j, i, j, p) -= 1.0;
                  if ((total – ind) == l)
                      break;
              }
          }
       } else if (total < l) {
          int ind = 0;
          for (i = 0; i < p; i++) {
              ind++;
              H(i, j, i, j, p) += 1.0;
              if ((total + ind) == l)
                 break;
          }
       }
   }
   total = 0.0;
   for (i = 0; i < p; i++) {
       double temp = ((double)csum[i]/(double)tsum)*l;
       if (temp < 1.0)
```

```
            temp = 1.0;
        temp = floor(temp);
        w[i] = temp;
    }
    for (i = 0; i < p; i++)
        total += w[i];
    if (total > l) {
       int ind = 0;
       for (i = 0; i < p; i++) {
           if (w[i] > 1.0) {
               ind++;
               w[i] -= 1.0;
               if ((total – ind) == l)
                  break;
           }
       }
    } else if (total < l) {
       int ind = 0;
       for (i = 0; i < p; i++) {
           ind++;
           w[i] += 1.0;
           if ((total + ind) == l)
               break;
       }
    }
    for (i = 0; i < p; i++) {
        for (j = 0; j < p; j++) {
           trow[i*p+j] = 0;
           for (k = 0; k < i; k++) {
              trow[i*p+j] += H(k, j, k, j, p);
           }
        }
    }
    for (i = 0; i < p; i++) {
        tcolumn[i] = 0;
        for (x = 0; x < i; x++) {
            tcolumn[i] += w[x];
        }
    }
    for (i = 0; i < p; i++) {
        for (j = 0; j < p; j++) {
            for (x = 0; x < p; x++) {
                for (y = 0; y < p; y++) {
                    int height = CommonHeight(
                                        trow[i*p+j],
                                        trow[i*p+j]+H(i, j, i, j, p),
                                        trow[x*p+y],
                                        trow[x*p+y]+H(x, y, x, y, p)
                    );
                    if (height > 0)
                      H(i, j, x, y, p) = height;
                }
            }
        }
    }
    return 0;
}
```

3. Steps to run the application

Outlined below are the steps to run the application:

1). Create the necessary VPM by using the command **mpccreate**. VPM is opened after successful execution of the command.

2). Compile the file

```
>mpcc –I$MPIDIR/include mxm.mpc
```

3). Copy the file ‘`mxm.c`’ produced after compilation into the $MPCLOAD directory on the host workstation.

```
>cp mxm.c $MPCLOAD/
```

4). Broadcast the file ‘`mxm.c`’ necessary to produce the executable to $MPCLOAD directory on other workstations constituting the VPM using the **mpcbcast** command.

```
>mpcbcast mxm.c
```

5). Run the executable using the **mpcrun** command.

```
>mpcrun mxm 3 2 36
```

6). The output from the execution would be:

```
Problem size=108, time(sec)=0.262353, time(min)=0.004373
```