Protokoll zum Praktikum Parallelrechner
Übung 4

Fakultät Informatik
TU Dresden

Christian Kroh
Matrikelnummer: 3755154
Studiengang: Informatik (Diplom)
Jahrgang: 2011/2012

10. Februar 2014, Dresden

Inhaltsverzeichnis

1 Matrizen-Multiplikation mit MPI

Submatrizen Die Ergebnis Matrix wird in Submatrizen unterteilt, die jeweils durch einen eigenen Prozess berechnet werden. Diese Blöcke umfassen jeweils `SUBDIMX` $*$ `SUBDIMY` Elemente.

Listing 1: matmul1-mpi.c - Matrix-Dimension und X- bzw. Y-Blöcke

```c
#define DIM 4096
#define PX 4
#define PY 4
#define SUBXDIM 1024
#define SUBYDIM 1024
```

1.1 Implementierung

Gruppen Jeder Prozess der Matrizenberechnung wird zwei MPI-Gruppen - einer Gruppe-X und einer Gruppe-Y, die angibt welchen Teil der B-Matrix bzw. der A-Matrix der Prozess benötigt - zugeteilt.

Listing 2: matmul1-mpi.c - X-Gruppen

```c
    MPI_Group_incl(orig_group, k, ranks, &(groupsx[i]));
    MPI_Comm_create(MPI_COMM_WORLD, groupsx[i], &(commsx[i]));
```

Listing 3: matmul1-mpi.c - Y-Gruppen

```c
    MPI_Group_incl(orig_group, k, ranks, &(groupsy[i]));
    MPI_Comm_create(MPI_COMM_WORLD, groupsy[i], &(commsy[i]));
```

Root-Prozess

- Initialisiert die Matrizen A und B mit zufälligen Werten.
- Generiert Submatrizen von A und B, in Abhängigkeit von `PY` bzw. `PX`.
- verteilt Submatrizen von A und B an die richtigen Prozesse
- Sammelt Ergebnisse von anderen Prozessen ein
- Berechnet eigenen Anteil der Ergebnismatrix

Listing 4: matmul1-mpi.c - Initialisierung von Matrizen A und B

```c
    int* A = random_mat( DIM );
    int* B = random_mat( DIM );
```

Listing 5: matmul1-mpi.c - Bestimmen der Submatrizen von A

```c
/*  create a-submatrixes*/
    for(int i = 0; i<PY; i++){
/*   copy rows of block i from matrix a to processes of group_y[i]    */
      for(int j = 0; j<SUBYDIM; j++){
        for(int k = 0; k<DIM; k++){
          a_submatrix[i][k + j * DIM] = A[k + (j + i * SUBYDIM) * DIM];
        }
      }
    }
```

Listing 6: matmul1-mpi.c - Bestimmen der Submatrizen von B

```c
/*  create b-submatrixes*/
    for(int i = 0; i<PX; i++){
/*   copy columns of block i from matrix b to processes of group_x[i]    */
      for(int k = 0; k<DIM; k++){
        for(int j = 0; j<SUBXDIM; j++){
          b_submatrix[i][j + k * SUBXDIM] = B[(i * SUBXDIM + j) + k * DIM];
        }
      }
    }
```

Listing 7: matmul1-mpi.c - Verteilen der Submatrizen von A an die jeweiligen Y-Gruppen

```
for(int i = 0; i<PY; i++){
  MPI_Bcast (a_submatrix[i], DIM * SUBYDIM, MPI_INT, 0, commsy[i]);
}
```

Listing 8: matmul1-mpi.c - Verteilen der Submatrizen von B an die jeweiligen X-Gruppen

```
for(int i = 0; i<PX; i++){
  MPI_Bcast (b_submatrix[i], DIM * SUBXDIM, MPI_INT, 0, commsx[i]);
}
```

Listing 9: matmul1-mpi.c - Einsammeln der Ergebnisse

```
int** C = ( int** )malloc( sizeof( int* ) * PY);
for(int i = 0; i<PY; i++){
  C[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
  MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, C[i], SUBXDIM * SUBYDIM, MPI_INT, 0,
       commsy[i]);
}
```

Berechnung von Teil-Ergebnismatrizen Alle Prozesse berechnen einen Anteil der Ergebnismatrix und senden ihr Ergebnis zurück an den Root-Prozess.

Listing 10: matmul1-mpi.c - Gruppen-Ids und weitergabe der A. bzw. B-Teilmatrizen an Rest der jeweiligen Gruppe

```
int grank_x, grank_y, groupx_id, groupy_id;

groupx_id = (rank) % PX;
groupy_id = (int) floor(rank / PX);
MPI_Group_rank(groupsx[groupx_id],&grank_x);
MPI_Group_rank(groupsy[groupy_id],&grank_y);

if(rank != 0){
  MPI_Bcast (a_submatrix[groupy_id], DIM * SUBYDIM, MPI_INT, 0, commsy[groupy_id]);
  MPI_Bcast (b_submatrix[groupx_id], DIM * SUBXDIM, MPI_INT, 0, commsx[groupx_id]);
 }
```

Listing 11: matmul1-mpi.c - Berechnung der Teil-Ergebnismatrix

```
/* Begin matrix matrix multiply kernel */
for ( uint32_t i = 0; i < SUBYDIM; i++ )
{
  for ( uint32_t k = 0; k < DIM; k++ )
  {
    for ( uint32_t j = 0; j < SUBXDIM; j++ )
    {
        // C[i][j] += A[i][k] * B[k][j]
         c_part[ i * SUBXDIM + j ] += a_submatrix[groupy_id][ i * DIM + k ] *
              b_submatrix[groupx_id][ k * DIM + j ];
    }
  }
}
  /* End matrix matrix multiply kernel */
```

Listing 12: matmul1-mpi.c - Berechnete Teil-Ergebnismatrix an den Root-Prozess zurücksenden

```
MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, c_part_return, SUBXDIM * SUBYDIM,
    MPI_INT, 0, commsy[groupy_id]);
```

2 Zeitmessungen

Gemessen mit Intel(R) Xeon(R) CPU E5-2690 (8 cores) @ 2.90GHz (Taurus).

Taurus: Matrix-Multiplikation Dimension 2048 x 2048

Prozesse	Runtime	S_p	GFLOP/s
1	7,476s	1	2,30
2	3,81s	1,988	4,51
4	2,1129s	3,585	8,13
8	1,0937s	6,926	15,71
16	0,5848s	12,9548	29,38

Taurus: Matrix-Multiplikation Dimension 4096 x 4096

Prozesse	Runtime	S_p	GFLOP/s
1	63,288s	1	2,17
2	31,2772s	2,0234	4,39
4	16,6630s	3,7981	8,25
8	8,96925s	7,0561	15,32
16	4,3631s	14,5052	31,50

3 Ergebnisse

3.1 Speedup

Der Speedup wächst abhängig von der Anzahl der verwendeten Prozesse zur Berechnung der Ergebnismatrix.

4 Anhang

4.1 Quellcode

Listing 13: matmul1-mpi.c - Quellcode

```

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <math.h>
#include <sys/time.h>
#include <math.h>
#include <x86intrin.h>
#include <mpi.h>

#define DIM 4096
#define PX 4
#define PY 4
#define SUBXDIM 1024
#define SUBYDIM 1024


static inline double gtod();
static inline int* random_mat( uint32_t n );
static inline int* zero_mat( uint32_t n );
static inline int* zero_mat_diff( uint32_t n, uint32_t m );


int main( int argc, char** argv )
{
    double t_start, t_end;
    double gflops;

  int** a_submatrix = ( int** )malloc( sizeof( int* ) * PY);
  int** b_submatrix = ( int** )malloc( sizeof( int* ) * PX);
  int* c_part = zero_mat_diff((DIM/PY), (DIM/PX));
  int* c_part_return;



  int rank, new_rank;
  int numtasks;

  MPI_Status status;

    MPI_Group orig_group;
  MPI_Group* groupsx = ( MPI_Group* )malloc( sizeof( MPI_Group ) * PX );
  MPI_Group* groupsy = ( MPI_Group* )malloc( sizeof( MPI_Group ) * PY );
  MPI_Comm* commsx = ( MPI_Comm* )malloc( sizeof( MPI_Comm ) * PX );
  MPI_Comm* commsy = ( MPI_Comm* )malloc( sizeof( MPI_Comm ) * PY );

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  if (numtasks != (PX * PY)) {
    printf("Must specify MP_PROCS= %d. Terminating.\n", (PX * PY));
    MPI_Finalize();
    exit(0);
  }

  MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

  int* ranks;
  int k;
  for(int i = 0; i<PX; i++){
```

```
  if(i == 0){
   ranks = ( int* )malloc( sizeof( int ) * (PY));
   k = 0;

  }else{
   ranks = ( int* )malloc( sizeof( int ) * (PY+1));
   k = 1;
   ranks[0]=0;
  }
  for(int j = i; j <= (i + PX * (PY-1)); j+=PX){
   ranks[k]=j;
/* if(rank ==0) printf("Process %3d: x-group %2d: %4d\n", rank, i, j);*/
   k++;
  }
  MPI_Group_incl(orig_group, k, ranks, &(groupsx[i]));
  MPI_Comm_create(MPI_COMM_WORLD, groupsx[i], &(commsx[i]));
  free(ranks);
  b_submatrix[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBXDIM);
 }

 for(int i = 0; i<PY; i++){
  if(i == 0){
   ranks = ( int* )malloc( sizeof( int ) * (PX));
   k = 0;
  }else{
   ranks = ( int* )malloc( sizeof( int ) * (PX+1));
   k = 1;
   ranks[0]=0;
  }
  for(int j = (i * PX); j < ((1 + i) * PX); j+=1){
/* if(rank ==0) printf("Process %3d: y-group %2d: %4d\n", rank, i, j);*/
   ranks[k]=j;
   k++;
  }
  MPI_Group_incl(orig_group, k, ranks, &(groupsy[i]));
  MPI_Comm_create(MPI_COMM_WORLD, groupsy[i], &(commsy[i]));
  free(ranks);
  a_submatrix[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
 }

 if (rank == 0)
 /* code for process zero */
 {

  int* A = random_mat( DIM );
  int* B = random_mat( DIM );

  if ( A == NULL || B == NULL)
  {
    printf( "Allocation of matrix failed.\n" );
    exit( EXIT_FAILURE );
  }

/*  create b-submatrixes*/
  for(int i = 0; i<PX; i++){
/* copy columns of block i from matrix b to processes of group_x[i]    */
   for(int k = 0; k<DIM; k++){
    for(int j = 0; j<SUBXDIM; j++){
     b_submatrix[i][j + k * SUBXDIM] = B[(i * SUBXDIM + j) + k * DIM];
    }
   }
  }

/*  create a-submatrixes*/
  for(int i = 0; i<PY; i++){
/* copy rows of block i from matrix a to processes of group_y[i]    */
   for(int j = 0; j<SUBYDIM; j++){
```

```
        for(int k = 0; k<DIM; k++){
          a_submatrix[i][k + j * DIM] = A[k + (j + i * SUBYDIM) * DIM];
        }
      }
    }


/*  broadcast submatrixes to groups*/

    for(int i = 0; i<PY; i++){
      MPI_Bcast (a_submatrix[i], DIM * SUBYDIM, MPI_INT, 0, commsy[i]);
    }

    for(int i = 0; i<PX; i++){
      MPI_Bcast (b_submatrix[i], DIM * SUBXDIM, MPI_INT, 0, commsx[i]);
    }

        free( A );
        free( B );



  }
  /* code for process one */

  int grank_x, grank_y, groupx_id, groupy_id;

  groupx_id = (rank) % PX;
  groupy_id = (int) floor(rank / PX);
  MPI_Group_rank(groupsx[groupx_id],&grank_x);
  MPI_Group_rank(groupsy[groupy_id],&grank_y);

  if(rank != 0){
    MPI_Bcast (a_submatrix[groupy_id], DIM * SUBYDIM, MPI_INT, 0, commsy[groupy_id]);
    MPI_Bcast (b_submatrix[groupx_id], DIM * SUBXDIM, MPI_INT, 0, commsx[groupx_id]);
   }


    t_start = gtod();

  /* Begin matrix matrix multiply kernel */
  for ( uint32_t i = 0; i < SUBYDIM; i++ )
  {
    for ( uint32_t k = 0; k < DIM; k++ )
    {
      for ( uint32_t j = 0; j < SUBXDIM; j++ )
      {
          // C[i][j] += A[i][k] * B[k][j]
           c_part[ i * SUBXDIM + j ] += a_submatrix[groupy_id][ i * DIM + k ] *
               b_submatrix[groupx_id][ k * DIM + j ];
      }
    }
  }
    /* End matrix matrix multiply kernel */

    t_end = gtod();
    gflops = ( ( double )2 * SUBXDIM * SUBYDIM * DIM / 1000000000.0 ) / ( t_end - t_start );

    printf("Process %3d worked ... Dim: %4d runtime: %7.4fs GFLOP/s: %0.2f\n", rank, DIM,
         t_end - t_start, gflops );

  if(rank != 0){
    MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, c_part_return, SUBXDIM * SUBYDIM,
         MPI_INT, 0, commsy[groupy_id]);
  }
```

```
  if(rank == 0){

    int** C = ( int** )malloc( sizeof( int* ) * PY);
    for(int i = 0; i<PY; i++){
      C[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
      MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, C[i], SUBXDIM * SUBYDIM, MPI_INT, 0, commsy[i]);
    }

    t_end = gtod();
    gflops = ( ( double )2 * (DIM) * (DIM) * DIM / 1000000000.0 ) / ( t_end - t_start );
      printf("Completed all in ... Dim: %4d runtime: %7.4fs GFLOP/s: %0.2f\n", DIM, t_end - t_start, gflops );

    for(int i=0; i<PY; i++){
      for(int j=0; j<(DIM/PY); j++){
        for(int k=0; k<DIM; k++){
          if(C[i][k + DIM * j] == (int) 0) printf("C[%d3][%3d + %5d * %3d] is NULL", i, k,DIM, j);
        }
      }
    }
  }
  MPI_Finalize();
    return EXIT_SUCCESS;
}


/** @brief Get current time stamp in seconds.
 *
 * @return Returns current time stamp in seconds.
 */
static inline double gtod( )
{
    struct timeval act_time;
    gettimeofday( &act_time, NULL );

    return ( double )act_time.tv_sec + ( double )act_time.tv_usec / 1000000.0;
}



/** @brief Generate randomized matrix.
 *
 * @param dim Dimension for the generated matrix.
 *
 * @return Returns a pointer to the generated matrix on success, NULL
 * otherwise.
 */
static inline int* random_mat( uint32_t dim )
{
    int *matrix = ( int* )malloc( sizeof( int ) * dim * dim );
    if ( matrix == NULL )
    {
        return NULL;
    }

    srand( ( unsigned ) time( NULL ) );

    for ( uint32_t i = 0; i < dim * dim; ++i)
    {
        matrix[ i ] = ( int )rand();
    }

  return matrix;
}

```

```c

/** @brief Generate zero matrix.
 *
 * @param dim Dimension for the generated matrix.
 *
 * @return Returns a pointer to the generated matrix on success, NULL
 * otherwise.
 */
static inline int* zero_mat( uint32_t dim )
{
    int* matrix = ( int* )malloc( sizeof( int ) * dim * dim );
    if ( matrix == NULL )
    {
        return NULL;
    }

    for ( uint32_t i = 0; i < dim * dim; ++i)
    {
        matrix[ i ] = ( int )0.0;
    }

  return matrix;
}



/** @brief Generate zero matrix.
 *
 * @param dim Dimension for the generated matrix.
 *
 * @return Returns a pointer to the generated matrix on success, NULL
 * otherwise.
 */
static inline int* zero_mat_diff( uint32_t dimx, uint32_t dimy )
{
    int* matrix = ( int* )malloc( sizeof( int ) * dimx * dimy );
    if ( matrix == NULL )
    {
        return NULL;
    }

    for ( uint32_t i = 0; i < dimx * dimy; ++i)
    {
        matrix[ i ] = ( int )0.0;
    }

  return matrix;
}
```