Protokoll zum Praktikum Parallelrechner
Übung 4

Fakultät Informatik
TU Dresden

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

12. 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

```
12 #define PX 4
13 #define PY 4
14 #define SUBXDIM 256
15 #define SUBYDIM 256
```

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

```
77     MPI_Comm_create(MPI_COMM_WORLD, groupsx[i], &(commsx[i]));
78     free(ranks);
```

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

```
97     MPI_Comm_create(MPI_COMM_WORLD, groupsy[i], &(commsy[i]));
98     free(ranks);
```

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

```
107    int* B = random_mat( DIM );
```

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

```
126    for(int i = 0; i<PY; i++){
127 /*   copy rows of block i from matrix a to processes of group_y[i]    */
128      for(int j = 0; j<SUBYDIM; j++){
129        for(int k = 0; k<DIM; k++){
130          a_submatrix[i][k + j * DIM] = A[k + (j + i * SUBYDIM) * DIM];
131        }
132      }
133    }
```

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

```
116    for(int i = 0; i<PX; i++){
117 /*   copy columns of block i from matrix b to processes of group_x[i]    */
118      for(int k = 0; k<DIM; k++){
119        for(int j = 0; j<SUBXDIM; j++){
120          b_submatrix[i][j + k * SUBXDIM] = B[(i * SUBXDIM + j) + k * DIM];
121        }
122      }
123    }
```

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

```
139       MPI_Bcast (a_submatrix[i], DIM * SUBYDIM, MPI_INT, 0, commsy[i]);
140     }
```

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

```
143       MPI_Bcast (b_submatrix[i], DIM * SUBXDIM, MPI_INT, 0, commsx[i]);
144     }
```

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

```
196     for(int i = 0; i<PY; i++){
197       C[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
198       MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, C[i], SUBXDIM * SUBYDIM, MPI_INT, 0,
              commsy[i]);
199     }
```

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

```
155
156   groupx_id = (rank) % PX;
157   groupy_id = (int) floor(rank / PX);
158   MPI_Group_rank(groupsx[groupx_id],&grank_x);
159   MPI_Group_rank(groupsy[groupy_id],&grank_y);
160
161   if(rank != 0){
162     MPI_Bcast (a_submatrix[groupy_id], DIM * SUBYDIM, MPI_INT, 0, commsy[groupy_id]);
163     MPI_Bcast (b_submatrix[groupx_id], DIM * SUBXDIM, MPI_INT, 0, commsx[groupx_id]);
164    }
```

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

```
170   for ( uint32_t i = 0; i < SUBYDIM; i++ )
171   {
172     for ( uint32_t k = 0; k < DIM; k++ )
173     {
174       for ( uint32_t j = 0; j < SUBXDIM; j++ )
175       {
176           // C[i][j] += A[i][k] * B[k][j]
177           c_part[ i * SUBXDIM + j ] += a_submatrix[groupy_id][ i * DIM + k ] *
                  b_submatrix[groupx_id][ k * DIM + j ];
178       }
179     }
180   }
181     /* End matrix matrix multiply kernel */
```

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

```
190   }
```

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 linear abhängig von der Anzahl der verwendeten Prozesse zur Berechnung der Ergebnismatrix, da die Rechenlast auf mehrere CPUs verteilt wird, die parallel zueinander arbeiten, wird das Ergebnis mit steigender Prozessanzahl schneller ermittelt.

3.2 Vergleich zur sequentiellen Berechnung

Durch die Aufteilung der Aufgabe in mehrere Einzelaufgaben, die parallel abgearbeitet werden, ist die Multiplikation von Matrizen unter Verwendung des MPI deutlich schneller als die sequentielle Berechnung aus Übung 1.

4 Anhang

4.1 Quellcode

Listing 13: matmul1-mpi.c - Quellcode

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <time.h>
5  #include <math.h>
6  #include <sys/time.h>
7  #include <math.h>
8  #include <x86intrin.h>
9  #include <mpi.h>
10
11 #define DIM 1024
12 #define PX 4
13 #define PY 4
14 #define SUBXDIM 256
15 #define SUBYDIM 256
16
17
18 static inline double gtod();
19 static inline int* random_mat( uint32_t n );
20 static inline int* zero_mat( uint32_t n );
21 static inline int* zero_mat_diff( uint32_t n, uint32_t m );
22
23
24 int main( int argc, char** argv )
25 {
26     double t_start, t_end;
27     double gflops;
28
29   int** a_submatrix = ( int** )malloc( sizeof( int* ) * PY);
30   int** b_submatrix = ( int** )malloc( sizeof( int* ) * PX);
31   int* c_part = zero_mat_diff((DIM/PY), (DIM/PX));
32   int* c_part_return;
33
34
35
36   int rank, new_rank;
37   int numtasks;
38
39   MPI_Status status;
40
41     MPI_Group orig_group;
42   MPI_Group* groupsx = ( MPI_Group* )malloc( sizeof( MPI_Group ) * PX );
43   MPI_Group* groupsy = ( MPI_Group* )malloc( sizeof( MPI_Group ) * PY );
44   MPI_Comm* commsx = ( MPI_Comm* )malloc( sizeof( MPI_Comm ) * PX );
45   MPI_Comm* commsy = ( MPI_Comm* )malloc( sizeof( MPI_Comm ) * PY );
46
47   MPI_Init( &argc, &argv );
48   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
49   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
50
51   if (numtasks != (PX * PY)) {
52     printf("Must specify MP_PROCS= %d. Terminating.\n", (PX * PY));
53     MPI_Finalize();
54     exit(0);
55   }
56
57   MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
58
59   int* ranks;
60   int k;
61   for(int i = 0; i<PX; i++){
62     if(i == 0){
```

```
63        ranks = ( int* )malloc( sizeof( int ) * (PY));
64        k = 0;
65
66      }else{
67        ranks = ( int* )malloc( sizeof( int ) * (PY+1));
68        k = 1;
69        ranks[0]=0;
70      }
71      for(int j = i; j <= (i + PX * (PY-1)); j+=PX){
72        ranks[k]=j;
73  /*    if(rank ==0) printf("Process %3d: x-group %2d: %4d\n", rank, i, j);*/
74        k++;
75      }
76      MPI_Group_incl(orig_group, k, ranks, &(groupsx[i]));
77      MPI_Comm_create(MPI_COMM_WORLD, groupsx[i], &(commsx[i]));
78      free(ranks);
79      b_submatrix[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBXDIM);
80    }
81
82    for(int i = 0; i<PY; i++){
83      if(i == 0){
84        ranks = ( int* )malloc( sizeof( int ) * (PX));
85        k = 0;
86      }else{
87        ranks = ( int* )malloc( sizeof( int ) * (PX+1));
88        k = 1;
89        ranks[0]=0;
90      }
91      for(int j = (i * PX); j < ((1 + i) * PX); j+=1){
92  /*    if(rank ==0) printf("Process %3d: y-group %2d: %4d\n", rank, i, j);*/
93        ranks[k]=j;
94        k++;
95      }
96      MPI_Group_incl(orig_group, k, ranks, &(groupsy[i]));
97      MPI_Comm_create(MPI_COMM_WORLD, groupsy[i], &(commsy[i]));
98      free(ranks);
99      a_submatrix[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
100   }
101
102   if (rank == 0)
103   /* code for process zero */
104   {
105
106     int* A = random_mat( DIM );
107     int* B = random_mat( DIM );
108
109     if ( A == NULL || B == NULL)
110     {
111         printf( "Allocation of matrix failed.\n" );
112         exit( EXIT_FAILURE );
113     }
114
115 /*  create b-submatrixes*/
116     for(int i = 0; i<PX; i++){
117 /*    copy columns of block i from matrix b to processes of group_x[i]    */
118       for(int k = 0; k<DIM; k++){
119         for(int j = 0; j<SUBXDIM; j++){
120           b_submatrix[i][j + k * SUBXDIM] = B[(i * SUBXDIM + j) + k * DIM];
121         }
122       }
123     }
124
125 /*  create a-submatrixes*/
126     for(int i = 0; i<PY; i++){
127 /*    copy rows of block i from matrix a to processes of group_y[i]    */
128       for(int j = 0; j<SUBYDIM; j++){
129         for(int k = 0; k<DIM; k++){
```

```
130           a_submatrix[i][k + j * DIM] = A[k + (j + i * SUBYDIM) * DIM];
131         }
132       }
133     }
134
135
136 /*  broadcast submatrixes to groups*/
137
138     for(int i = 0; i<PY; i++){
139       MPI_Bcast (a_submatrix[i], DIM * SUBYDIM, MPI_INT, 0, commsy[i]);
140     }
141
142     for(int i = 0; i<PX; i++){
143       MPI_Bcast (b_submatrix[i], DIM * SUBXDIM, MPI_INT, 0, commsx[i]);
144     }
145
146         free( A );
147         free( B );
148
149
150
151   }
152   /* code for process one */
153
154   int grank_x, grank_y, groupx_id, groupy_id;
155
156   groupx_id = (rank) % PX;
157   groupy_id = (int) floor(rank / PX);
158   MPI_Group_rank(groupsx[groupx_id],&grank_x);
159   MPI_Group_rank(groupsy[groupy_id],&grank_y);
160
161   if(rank != 0){
162     MPI_Bcast (a_submatrix[groupy_id], DIM * SUBYDIM, MPI_INT, 0, commsy[groupy_id]);
163     MPI_Bcast (b_submatrix[groupx_id], DIM * SUBXDIM, MPI_INT, 0, commsx[groupx_id]);
164    }
165
166
167     t_start = gtod();
168
169   /* Begin matrix matrix multiply kernel */
170   for ( uint32_t i = 0; i < SUBYDIM; i++ )
171   {
172     for ( uint32_t k = 0; k < DIM; k++ )
173     {
174       for ( uint32_t j = 0; j < SUBXDIM; j++ )
175       {
176           // C[i][j] += A[i][k] * B[k][j]
177           c_part[ i * SUBXDIM + j ] += a_submatrix[groupy_id][ i * DIM + k ] *
                b_submatrix[groupx_id][ k * DIM + j ];
178       }
179     }
180   }
181     /* End matrix matrix multiply kernel */
182
183     t_end = gtod();
184     gflops = ( ( double )2 * SUBXDIM * SUBYDIM * DIM / 1000000000.0 ) / ( t_end - t_start );
185
186     printf("Process %3d worked ... Dim: %4d runtime: %7.4fs GFLOP/s: %0.2f\n", rank, DIM,
          t_end - t_start, gflops );
187
188   if(rank != 0){
189     MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, c_part_return, SUBXDIM * SUBYDIM,
          MPI_INT, 0, commsy[groupy_id]);
190   }
191
192
193   if(rank == 0){
```

```
194
195     int** C = ( int** )malloc( sizeof( int* ) * PY);
196     for(int i = 0; i<PY; i++){
197       C[i] = ( int* )malloc( sizeof( MPI_INT ) * DIM * SUBYDIM);
198       MPI_Gather( c_part, SUBXDIM * SUBYDIM, MPI_INT, C[i], SUBXDIM * SUBYDIM, MPI_INT, 0,
              commsy[i]);
199     }
200
201     t_end = gtod();
202     gflops = ( ( double )2 * (DIM) * (DIM) * DIM / 1000000000.0 ) / ( t_end - t_start );
203       printf("Completed all in ... Dim: %4d runtime: %7.4fs GFLOP/s: %0.2f\n", DIM, t_end -
              t_start, gflops );
204
205     for(int i=0; i<PY; i++){
206       for(int j=0; j<(DIM/PY); j++){
207         for(int k=0; k<DIM; k++){
208           if(C[i][k + DIM * j] == (int) 0) printf("C[%d3][%3d + %5d * %3d] is NULL", i,
                  k,DIM, j);
209         }
210       }
211     }
212   }
213   MPI_Finalize();
214     return EXIT_SUCCESS;
215 }
216
217
218 /** @brief Get current time stamp in seconds.
219  *
220  * @return Returns current time stamp in seconds.
221  */
222 static inline double gtod( )
223 {
224     struct timeval act_time;
225     gettimeofday( &act_time, NULL );
226
227     return ( double )act_time.tv_sec + ( double )act_time.tv_usec / 1000000.0;
228 }
229
230
231
232 /** @brief Generate randomized matrix.
233  *
234  * @param dim Dimension for the generated matrix.
235  *
236  * @return Returns a pointer to the generated matrix on success, NULL
237  * otherwise.
238  */
239 static inline int* random_mat( uint32_t dim )
240 {
241     int *matrix = ( int* )malloc( sizeof( int ) * dim * dim );
242     if ( matrix == NULL )
243     {
244         return NULL;
245     }
246
247     srand( ( unsigned ) time( NULL ) );
248
249     for ( uint32_t i = 0; i < dim * dim; ++i)
250     {
251         matrix[ i ] = ( int )rand();
252     }
253
254   return matrix;
255 }
256
257
```

```
258 /** @brief Generate zero matrix.
259  *
260  * @param dim Dimension for the generated matrix.
261  *
262  * @return Returns a pointer to the generated matrix on success, NULL
263  * otherwise.
264  */
265 static inline int* zero_mat( uint32_t dim )
266 {
267     int* matrix = ( int* )malloc( sizeof( int ) * dim * dim );
268     if ( matrix == NULL )
269     {
270         return NULL;
271     }
272 
273     for ( uint32_t i = 0; i < dim * dim; ++i)
274     {
275         matrix[ i ] = ( int )0.0;
276     }
277 
278   return matrix;
279 }
280 
281 
282 
283 /** @brief Generate zero matrix.
284  *
285  * @param dim Dimension for the generated matrix.
286  *
287  * @return Returns a pointer to the generated matrix on success, NULL
288  * otherwise.
289  */
290 static inline int* zero_mat_diff( uint32_t dimx, uint32_t dimy )
291 {
292     int* matrix = ( int* )malloc( sizeof( int ) * dimx * dimy );
293     if ( matrix == NULL )
294     {
295         return NULL;
296     }
297 
298     for ( uint32_t i = 0; i < dimx * dimy; ++i)
299     {
300         matrix[ i ] = ( int )0.0;
301     }
302 
303   return matrix;
304 }
```