

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΜ&ΜΥ Συστήματα Παράλληλης Επεξεργασίας 1^η Άσκηση Ακ. έτος 2012-2013

Ομάδα 8η

Μαυρογιάννης Αλέξανδρος Α.Μ.: 03109677 Λύρας Γρηγόρης Α.Μ.: 03109687

Πηγαίος κώδικας

Κοινή βιβλιοθήκη

```
/* -.-.-.-.-.
     * File Name : common.h
2
     * Creation Date : 06-11-2012
3
     * Last Modified : Sun 11 Nov 2012 06:12:55 PM EET
     * Created By : Greg Liras <gregliras@gmail.com>
5
     _-----*/
     #ifndef DEBUG FUNC
     #define DEBUG_FUNC
     \#if\ main\_DEBUG
11
     #define debug(fmt,arg...)
                                fprintf(stdout, "%s: " fmt, __func__ , ##arg)
13
                                do { } while(0)
     #define debug(fmt,arg...)
14
15
     #endif /* main_DEBUG */
16
     #endif /* DEBUG_FUNC */
18
     #ifndef COMMON H
19
     #define COMMON_H
21
22
23
     #include <stdlib.h>
24
     #include <stdio.h>
25
     #include <mpi.h>
27
     double *allocate_2d(int N, int M);
     double *allocate_2d_with_padding(int N, int M, int max_rank);
     double *parse_matrix_2d(FILE *fp, int N, int M, double *A);
30
     void fprint_matrix_2d(FILE *fp, int N, int M, double *A);
31
     void print_matrix_2d(int N, int M, double *A);
32
     double timer(void);
34
     void usage(int argc, char **argv);
35
     #ifdef USE_MPI /* USE_MPI */
     void propagate_with_send(void *buffer, int count , MPI_Datatype datatype, int root, MPI_Comm comm);
37
     void propagate_with_flooding(void *buffer, int count , MPI_Datatype datatype, int root, MPI_Comm comm);
38
     #endif /* USE_MPI */
40
     #endif /* COMMON_H */
41
    /* -.-.-.-.-.
     * File Name : common.c
     * Creation Date : 06-11-2012
     * Last Modified : Sun 11 Nov 2012 06:40:49 PM EET
     * Created By : Greg Liras <gregliras@gmail.com>
     #include "common.h"
     #include <sys/time.h>
     double *allocate_2d(int N, int M)
11
12
13
        A = malloc(N * M * sizeof(double));
14
        return A;
15
16
17
     double *allocate_2d_with_padding(int N, int M, int max_rank)
18
19
         double *A;
         A = allocate_2d(N + max_rank, M);
21
        return A:
22
     }
24
25
     double *parse_matrix_2d(FILE *fp, int N, int M, double *A)
27
     {
28
         int i,j;
        double *p;
```

```
p = A;
30
          for (i = 0; i < N; i++) {
31
              for (j = 0; j < M; j++) {
    if(!fscanf(fp, "%lf", p++)) {</pre>
32
33
                       return NULL;
34
35
              }
36
          }
37
38
          return A;
     }
39
40
     void fprint_matrix_2d(FILE *fp, int N, int M, double *A)
41
42
43
          int i,j;
          double *p;
44
          p = A;
45
46
          for (j = 0; j < M; j++) {
             fprintf(fp, "=");
47
48
          fprintf(fp, "\n");
49
          for (i = 0; i < N; i++) {
50
              for (j = 0; j < M; j++) {
                  fprintf(fp, "%lf\t", *p++);
52
53
              fprintf(fp, "\n");
54
55
          for (j = 0; j < M; j++) {
56
              fprintf(fp, "=");
57
58
59
          fprintf(fp, "\n");
     }
60
61
     void print_matrix_2d(int N, int M, double *A)
62
63
64
          fprint_matrix_2d(stdout, N, M, A);
65
     }
66
67
     double timer(void)
68
69
70
          static double seconds = 0;
          static int operation = 0;
71
72
          struct timeval tv;
          gettimeofday(&tv, NULL);
73
          if (operation == 0) {
74
75
              seconds = tv.tv_sec + (((double) tv.tv_usec)/1e6);
              operation = 1;
76
              return 0;
77
78
          }
          else {
79
80
              operation = 0;
              return tv.tv_sec + (((double) tv.tv_usec)/1e6) - seconds;
81
82
     }
84
     void usage(int argc, char **argv)
85
86
          if(argc != 3) {
87
              printf("Usage: %s <matrix file> <output file>\n", argv[0]);
88
              exit(EXIT_FAILURE);
89
90
     }
91
92
     #ifdef USE_MPI /* USE_MPI */
93
94
     void propagate_with_send(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
95
96
          int rank;
97
          int i;
          int max_rank;
98
99
          MPI_Comm_rank(comm, &rank);
100
          MPI_Comm_size(comm, &max_rank);
101
          if(rank == root) {
102
              for(i = 0; i < max_rank; i++) {</pre>
103
                  if(i == rank) {
104
```

```
continue;
105
                   }
106
107
                   else {
                        debug("%d\n", i);
108
109
                        MPI_Send(buffer, count, datatype, i, root, comm);
110
              }
111
          }
112
          else {
113
               MPI_Status status;
114
115
               MPI_Recv(buffer, count, datatype, root, root, comm, &status);
116
117
118
119
      }
120
121
      void propagate_with_flooding(void *buffer, int count , MPI_Datatype datatype, int root, MPI_Comm comm)
122
123
          int rank;
124
          int max rank:
125
          int cur;
126
127
          MPI_Comm_rank(comm, &rank);
128
          MPI_Comm_size(comm, &max_rank);
129
130
          if(root != 0) {
131
               if(rank == root) {
132
                   MPI_Send(buffer, count, datatype, 0, root, comm);
133
134
               if(rank == 0) {
135
136
                   MPI_Status status;
                   MPI_Recv(buffer, count, datatype, root, root, comm, &status);
137
138
          }
139
140
141
142
          if(rank != 0) {
               MPI_Status status;
143
               MPI_Recv(buffer, count, datatype, (rank-1)/2, root, comm, &status);
144
145
          cur = 2*rank+1;
146
          if(cur < max_rank) {</pre>
147
               MPI_Send(buffer, count, datatype, cur, root, comm);
148
149
150
          if(++cur < max_rank) {</pre>
               MPI_Send(buffer, count, datatype, cur, root, comm);
151
152
153
     #endif /* USE MPI */
154
```

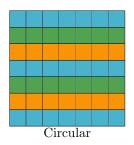
Ζητούμενο 1 Σειριακό Πρόγραμμα

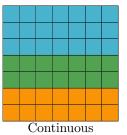
```
1
    * File Name : main.c
2
     * Creation Date : 30-10-2012
3
     * Last Modified : Thu 08 Nov 2012 09:50:55 AM EET
     * Created By : Greg Liras <gregliras@gmail.com>
     * Created By : Alex Maurogiannis <nalfemp@gmail.com>
     _-----*/
     #include <stdio.h>
10
    #include <stdlib.h>
11
    #include "common.h"
12
13
    int main(int argc, char **argv)
14
15
        int i,j,k;
16
        int N;
17
18
        double *A;
        double 1;
19
        double sec;
20
21
22
        FILE *fp = NULL;
```

```
23
         usage(argc, argv);
24
25
           * Allocate me!
26
         fp = fopen(argv[1], "r");
27
         if(fp) {
28
              if(!fscanf(fp, "%d\n", &N)) {
29
                  exit(EXIT_FAILURE);
30
31
         }
32
33
         if((A = allocate_2d(N, N)) == NULL) {
34
              exit(EXIT_FAILURE);
35
         if(parse_matrix_2d(fp, N, N, A) == NULL) {
37
              exit(EXIT_FAILURE);
38
39
40
41
         sec = timer();
42
         for (k = 0; k < N - 1; k++)
43
              for (i = k + 1; i < N; i++)
45
46
                  1 = A[i * N + k] / A[k * N + k];
47
48
                  for (j = k; j < N; j++)
49
                      A[i * N + j] = A[i * N + j] - 1 * A[k * N + j];
51
52
              }
53
54
         sec = timer();
         printf("Calc Time: %lf\n", sec);
55
56
57
         fp = fopen(argv[2], "w");
58
         fprint_matrix_2d(fp, N, N, A);
         fclose(fp);
59
         free(A);
61
         return 0;
62
    }
```

Ζητούμενο 2 Παραλληλισμός Αλγορίθμου

Ο παραλληλισμός του αλγορίθμου εντοπίζεται στο γεγονός ότι υπάρχει ανεξαρτησία του υπολογισμού κατά γραμμές για δεδομένο k. Καθ όλη την εκτέλεση του προγράμματος κρατάμε σταθερό τον τρόπο διαμοιρασμού των γραμμών του πίνακα, μοιράζοντας σε κάθε επανάληψη την k^{th} γραμμή.





Ζητούμενο 3 Μοντέλο κοινού χώρου διευθύνσεων (OpenMP)

Ζητούμενο 4 Μοντέλο ανταλλαγής μηνυμάτων (ΜΡΙ)

Ζητούμενο 4.1 Point to Point

```
#include <mpi.h>
     #include <stdio.h>
10
     #include <stdlib.h>
11
     #include <signal.h>
12
     #include <signal.h>
13
     #include <unistd.h>
14
     #include <string.h>
15
16
     #include "common.h"
17
18
19
     #define BLOCK_ROWS 1
20
21
22
     void process_rows(int k, int rank, int N, int max_rank, double *A){
                 performs the calculations for a given set of rows.
23
                  In this hybrid version each thread is assigned blocks of
24
25
                  continuous rows in a cyclic manner.
          */
26
27
         int i, j, w;
28
         double 1;
         /* For every cyclic repetition of a block */
29
         for (i = (rank + ((BLOCK_ROWS * max_rank) * (k / (BLOCK_ROWS * max_rank)))); i < N ; i+=(max_rank * BLOCK_ROWS)) {
                 if (i > k) {
31
                      /* Calculate each continuous row in the block*/
32
                      for (w = i; w < (i + BLOCK_ROWS) && w < (N * N); w++){
33
                          1 = A[(w * N) + k] / A[(k * N) + k];
34
35
                          for (j = k; j < N; j++) {
                              A[(w * N) + j] = A[(w * N) + j] - 1 * A[(k* N) + j];
36
37
38
                      }
                 }
39
40
         }
     }
41
42
43
     int main(int argc, char **argv)
44
45
         int k:
         int N;
47
         int rank;
48
         int max_rank;
         int last_rank;
49
         double *A = NULL;
50
51
         double sec = 0;
52
         int ret = 0;
53
54
         FILE *fp = NULL;
         usage(argc, argv);
55
56
57
         MPI_Init(&argc, &argv);
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
58
59
         MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
60
         if (rank == 0) {
61
             debug("rank: %d opens file: %s\n", rank, argv[1]);
             fp = fopen(argv[1], "r");
63
64
             if(fp) {
                  if(!fscanf(fp, "d\n", &N)) {
65
                      MPI_Abort(MPI_COMM_WORLD, 1);
66
67
             }
             else {
69
70
                 MPI_Abort(MPI_COMM_WORLD, 1);
71
72
         }
73
74
         MPI_Barrier(MPI_COMM_WORLD);
75
         propagate_with_flooding(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
76
77
78
         /* Everyone allocates the whole table */
         debug("Max rank = %d\n", max_rank);
79
         if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
80
             MPI_Abort(MPI_COMM_WORLD, 1);
82
         /* Root Parses file */
83
```

```
if (rank == 0) {
84
              if(parse_matrix_2d(fp, N, N, A) == NULL) {
85
                  MPI_Abort(MPI_COMM_WORLD, 1);
87
              fclose(fp);
88
              fp = NULL;
89
90
          /* And distributes the table */
91
          MPI_Barrier(MPI_COMM_WORLD);
92
         propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
93
94
          last_rank = (N - 1) % max_rank;
95
96
97
          if(rank == 0) {
              sec = timer();
98
99
100
          for (k = 0; k < N - 1; k++) {
101
102
              /* The owner of the row for this k broadcasts it*/
              MPI_Barrier(MPI_COMM_WORLD);
103
              propagate\_with\_flooding(\&A[k*N], N, MPI\_DOUBLE, ((k \% (max\_rank*BLOCK\_ROWS)) / BLOCK\_ROWS), MPI\_COMM\_WORLD);\\
104
105
              process_rows(k, rank, N, max_rank, A);
106
107
108
          MPI_Barrier(MPI_COMM_WORLD);
109
110
          if (rank == 0) {
              sec = timer();
111
              printf("Calc Time: %lf\n", sec);
112
113
         ret = MPI_Finalize();
114
115
          if(ret == 0) {
116
              debug("%d FINALIZED!!! with code: %d\n", rank, ret);
117
118
119
          else {
              debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
120
121
122
          /* Last process has table */
123
124
          if (rank == last_rank) {
              //print_matrix_2d(N, N, A);
fp = fopen(argv[2], "w");
125
126
              fprint_matrix_2d(fp, N, N, A);
127
128
              fclose(fp);
129
          free(A);
130
131
132
          return 0;
    }
133
     /* -.-.-.-.-.-.
      * File Name : main.c
       * Creation Date : 30-10-2012
 3
       * Last Modified : Sun 11 Nov 2012 09:03:29 PM EET
       * Created By : Greg Liras <gregliras@gmail.com>
      * Created By : Alex Maurogiannis <nalfemp@gmail.com>
 6
      _-----*/
     #include <mpi.h>
      #include <stdio.h>
10
     #include <stdlib.h>
11
12
     #include <siqnal.h>
      #include <signal.h>
13
     #include <unistd.h>
14
15
     #include <string.h>
16
     #include "common.h"
17
18
19
     int get_bcaster(int *ccounts, int bcaster) {
20
          if (ccounts[bcaster]-- > 0 ){
21
             return bcaster:
22
23
          } else {
              return bcaster+1;
24
25
```

```
|}
26
27
28
     void get_displs(int *counts, int max_rank, int *displs) {
29
          int j;
          displs[0] = 0;
30
          for (j = 1; j < max_rank; j++) {
31
              displs[j] = displs[j - 1] + counts[j - 1];
32
33
     }
34
35
36
     int max(int a, int b) {
         return a > b ? a : b;
37
38
39
     void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, double *A){
40
                 performs the calculations for a given set of rows.
41
42
                  In this hybrid version each thread is assigned blocks of
                  continuous rows in a cyclic manner.
43
          */
44
         int j, w;
45
         double 1:
46
          int start = max(displs[rank], k+1);
          for (w = start; w < (start + block_rows) && w < N; w++){
48
              1 = A[(w * N) + k] / A[(k * N) + k];
49
              for (j = k; j < N; j++) {
                  A[(w * N) + j] = A[(w * N) + j] - 1 * A[(k * N) + j];
51
52
         }
53
     }
54
55
     /* distributes the rows in a continuous fashion */
56
     void distribute_rows(int max_rank, int N, int *counts) {
57
              int j, k;
58
              int rows = N;
59
60
61
              /* Initialize counts */
             for (j = 0; j < max_rank ; j++) {
62
63
                  counts[j] = (rows / max_rank);
64
65
              /* Distribute the indivisible leftover */
              if (rows / max_rank != 0) {
67
                  j = rows % max_rank;
68
                  for (k = 0; k < max_rank && j > 0; k++, j--) {
                          counts[k] += 1;
70
71
                  }
              } else {
72
                  for (k = 0; k < max_rank; k++){
73
74
                      counts[k] = 1;
75
              }
76
77
     }
78
80
81
     int main(int argc, char **argv)
82
          int i, j, k;
83
          int N:
84
          int rank;
85
         int max_rank;
86
87
          int block_rows;
         int *counts;
88
89
         int *displs;
          int *ccounts;
          int ret = 0;
91
92
          int bcaster = 0;
93
          double 1;
         double sec;
94
95
          double *A = NULL;
         FILE *fp = NULL;
96
97
         usage(argc, argv);
99
100
```

```
MPI_Init(&argc, &argv);
101
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
102
103
          MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
104
          if (rank == 0) {
105
              debug("rank: %d opens file: %s\n", rank, argv[1]);
106
              fp = fopen(argv[1], "r");
107
              if(fp) {
108
                   if(!fscanf(fp, "%d\n", &N)) {
109
                       MPI_Abort(MPI_COMM_WORLD, 1);
110
111
              }
112
113
              else {
                   MPI_Abort(MPI_COMM_WORLD, 1);
114
115
116
117
118
119
          MPI_Barrier(MPI_COMM_WORLD);
          propagate_with_flooding(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
120
121
          counts = malloc(max_rank * sizeof(int));
122
          displs = malloc(max_rank * sizeof(int));
123
          ccounts = malloc(max_rank * sizeof(int));
124
125
126
          distribute_rows(max_rank, N, counts);
127
          get_displs(counts, max_rank, displs);
          memcpy(ccounts, counts, max_rank * sizeof(int));
128
      #if main_DEBUG
129
130
              printf("CCounts is :\n");
              for (j = 0; j < max_rank; j++) {
131
132
                  printf("%d\n", ccounts[j]);
133
      #endif
134
135
136
          /* Everybody Allocates the whole table */
          if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
137
              MPI_Abort(MPI_COMM_WORLD, 1);
138
139
          if (rank == 0) {
140
              if(parse_matrix_2d(fp, N, N, A) == NULL) {
141
                  MPI_Abort(MPI_COMM_WORLD, 1);
142
143
              fclose(fp);
144
              fp = NULL;
145
146
          }
147
          MPI_Barrier(MPI_COMM_WORLD);
148
149
          propagate_with_flooding(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
150
151
          /* Start Timing */
          if(rank == 0) {
152
              sec = timer();
153
154
155
156
          for (k = 0; k < N - 1; k++) {
157
              block_rows = counts[rank];
158
159
              bcaster = get_bcaster(ccounts, bcaster);
160
              debug(" broadcaster is %d\n", bcaster);
161
162
              MPI_Barrier(MPI_COMM_WORLD);
              propagate_with_flooding(&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD);
163
164
              process_rows(k, rank, N, max_rank, block_rows, displs, A);
165
166
167
          MPI_Barrier(MPI_COMM_WORLD);
168
          if(rank == 0) {
169
170
              sec = timer();
              printf("Calc Time: %lf\n", sec);
171
172
          ret = MPI_Finalize();
173
174
          if(ret == 0) {
175
```

```
debug("%d FINALIZED!!! with code: %d\n", rank, ret);
176
177
178
          else {
              debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
179
180
181
          if(rank == (max_rank - 1)) {
182
              fp = fopen(argv[2], "w");
183
              fprint_matrix_2d(fp, N, N, A);
184
             fclose(fp);
185
186
187
         free(A);
          free(counts):
188
189
          free(ccounts);
190
191
         return 0;
192
     }
     Ζητούμενο 4.2 Collective
      * File Name : main.c
 2
 3
      * Creation Date : 30-10-2012
      * Last Modified : Sun 11 Nov 2012 09:36:48 PM EET
      * Created By : Greg Liras <gregliras@gmail.com>
 5
      * Created By : Alex Maurogiannis <nalfemp@gmail.com>
       #include <mpi.h>
     #include <stdio.h>
10
11
     #include <stdlib.h>
      #include <siqnal.h>
     #include <signal.h>
13
14
     \#include < unistd.h >
      #include <string.h>
15
16
     \#include "common.h"
18
     #define BLOCK_ROWS 1
19
21
     void process_rows(int k, int rank, int N, int max_rank, double *A){
22
                performs the calculations for a given set of rows.
23
                 In this hybrid version each thread is assigned blocks of
24
25
                 continuous rows in a cyclic manner.
          */
26
27
         int i, j, w;
         double 1;
         /* For every cyclic repetition of a block */
29
         for (i = (rank + ((BLOCK_ROWS * max_rank) * (k / (BLOCK_ROWS * max_rank)))); i < N ; i+=(max_rank * BLOCK_ROWS)) {
30
31
                 if (i > k) {
                      /* Calculate each continuous row in the block*/
32
                      for (w = i; w < (i + BLOCK_ROWS) && w < (N * N); w++){}
33
                          1 = A[(w * N) + k] / A[(k * N) + k];
34
                          for (j = k; j < N; j++) {
35
                              A[(w * N) + j] = A[(w * N) + j] - 1 * A[(k* N) + j];
37
                      }
38
                 }
39
40
         }
     }
41
42
     int main(int argc, char **argv)
43
44
         int k;
45
46
         int N:
          int rank;
47
          int max rank;
48
          int last_rank;
         double *A = NULL;
50
         double sec = 0;
51
          int ret = 0;
53
         FILE *fp = NULL;
54
         usage(argc, argv);
```

```
56
          MPI_Init(&argc, &argv);
57
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
          MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
59
60
          if (rank == 0) {
61
              debug("rank: %d opens file: %s\n", rank, argv[1]);
62
              fp = fopen(argv[1], "r");
63
              if(fp) {
64
                  if(!fscanf(fp, "d\n", &N)) {
65
66
                       MPI_Abort(MPI_COMM_WORLD, 1);
67
              }
68
              else {
69
                  MPI_Abort(MPI_COMM_WORLD, 1);
70
71
72
73
74
          MPI_Barrier(MPI_COMM_WORLD);
75
          MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
76
77
          /* Everyone allocates the whole table */
78
          debug("Max rank = %d\n", max_rank);
79
          if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
80
              MPI_Abort(MPI_COMM_WORLD, 1);
81
82
          /* Root Parses file */
83
          if (rank == 0) {
84
              if(parse_matrix_2d(fp, N, N, A) == NULL) {
85
                  MPI_Abort(MPI_COMM_WORLD, 1);
86
87
              fclose(fp);
88
              fp = NULL;
89
91
          /* And distributes the table */
         MPI_Barrier(MPI_COMM_WORLD);
92
          MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
93
94
          last_rank = (N - 1) % max_rank;
95
          if(rank == 0) {
97
98
              sec = timer();
99
100
101
          for (k = 0; k < N - 1; k++) {
              /* The owner of the row for this k broadcasts it*/
102
              MPI_Barrier(MPI_COMM_WORLD);
103
104
              MPI_Bcast(&A[k * N], N, MPI_DOUBLE, ((k % (max_rank * BLOCK_ROWS)) / BLOCK_ROWS), MPI_COMM_WORLD);
105
106
              process_rows(k, rank, N, max_rank, A);
107
108
          MPI_Barrier(MPI_COMM_WORLD);
109
          if (rank == 0) {
110
              sec = timer();
111
              printf("Calc Time: %lf\n", sec);
112
113
          ret = MPI_Finalize();
114
115
          if(ret == 0) {
116
117
              debug("%d FINALIZED!!! with code: %d\n", rank, ret);
118
119
          else {
              debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
120
121
122
          /* Last process has table */
123
          if (rank == last_rank) {
124
125
              //print_matrix_2d(N, N, A);
              fp = fopen(argv[2], "w");
126
              fprint_matrix_2d(fp, N, N, A);
127
              fclose(fp);
128
129
130
          free(A);
```

```
131
         return 0:
132
    | }
133
     /* -.-.-.-.-.-.
      * File Name : main.c
2
 3
      * Creation Date : 30-10-2012
       * Last Modified : Sun 11 Nov 2012 09:03:29 PM EET
 4
      * Created By : Greg Liras \langle gregliras@gmail.com \rangle
 5
      * Created By : Alex Maurogiannis \langle nalfemp@gmail.com \rangle
       _------*/
     #include <mpi.h>
     #include <stdio.h>
10
11
     #include <stdlib.h>
     #include <signal.h>
12
     #include <signal.h>
13
     #include <unistd.h>
     #include <string.h>
15
     #include "common.h"
18
19
     int get_bcaster(int *ccounts, int bcaster) {
20
21
          if (ccounts[bcaster]-- > 0 ){
22
              return bcaster;
         } else {
23
24
              return bcaster+1;
25
     }
26
27
28
     void get_displs(int *counts, int max_rank, int *displs) {
29
          int j;
          displs[0] = 0;
          for (j = 1; j < max_rank ; j++) {</pre>
31
              displs[j] = displs[j - 1] + counts[j - 1];
32
33
     }
34
35
     int max(int a, int b) {
36
         return a > b ? a : b;
37
38
39
40
     void process_rows(int k, int rank, int N, int max_rank, int block_rows, int *displs, double *A){
41
                  performs the calculations for a given set of rows.
                  In this hybrid version each thread is assigned blocks of
42
43
                  continuous rows in a cyclic manner.
44
          */
         int j, w;
45
         double 1;
47
          int start = max(displs[rank], k+1);
          for (w = start; w < (start + block_rows) && w < N; w++){
48
             1 = A[(w * N) + k] / A[(k * N) + k];
             for (j = k; j < N; j++) {

A[(w * N) + j] = A[(w * N) + j] - 1 * A[(k * N) + j];
50
51
52
         }
53
     }
54
55
      /* distributes the rows in a continuous fashion */
56
57
     void distribute_rows(int max_rank, int N, int *counts) {
             int j, k;
58
             int rows = N;
59
60
              /* Initialize counts */
61
62
              for (j = 0; j < max_rank; j++) {
                  counts[j] = (rows / max_rank);
63
64
              /* Distribute the indivisible leftover */
66
              if (rows / max_rank != 0) {
67
                  j = rows % max_rank;
                  for (k = 0; k < max_rank && j > 0; k++, j--) {
69
70
                          counts[k] += 1;
                  }
71
             } else {
```

```
for (k = 0; k < max_rank; k++){
73
                       counts[k] = 1;
74
75
                   }
76
77
      }
78
79
80
      int main(int argc, char **argv)
81
82
83
          int i, j, k;
          int N;
84
          int rank:
85
          int max_rank;
          int block_rows;
87
88
          int *counts;
89
          int *displs;
          int *ccounts;
90
91
          int ret = 0;
          int bcaster = 0;
92
          double 1:
93
          double sec;
          double *A = NULL;
95
          FILE *fp = NULL;
96
97
98
99
          usage(argc, argv);
100
          MPI_Init(&argc, &argv);
101
102
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
          MPI_Comm_size(MPI_COMM_WORLD, &max_rank);
103
104
          if (rank == 0) {
105
              debug("rank: %d opens file: %s\n", rank, argv[1]);
106
107
              fp = fopen(argv[1], "r");
108
              if(fp) {
                   if(!fscanf(fp, "%d\n", &N)) {
109
110
                       MPI_Abort(MPI_COMM_WORLD, 1);
111
              }
112
113
                   MPI_Abort(MPI_COMM_WORLD, 1);
114
115
116
117
118
          MPI_Barrier(MPI_COMM_WORLD);
119
          MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
120
121
          counts = malloc(max_rank * sizeof(int));
122
123
          displs = malloc(max_rank * sizeof(int));
          ccounts = malloc(max_rank * sizeof(int));
124
125
126
          distribute_rows(max_rank, N, counts);
          get_displs(counts, max_rank, displs);
127
128
          memcpy(ccounts, counts, max_rank * sizeof(int));
      #if main_DEBUG
129
              printf("CCounts is :\n");
130
              for (j = 0; j < max_rank; j++) {
131
                   printf("%d\n", ccounts[j]);
132
133
      #endif
134
135
          /* Everybody Allocates the whole table */
136
137
          if((A = allocate_2d_with_padding(N, N, max_rank)) == NULL) {
              MPI_Abort(MPI_COMM_WORLD, 1);
138
139
          if (rank == 0) {
140
              if(parse_matrix_2d(fp, N, N, A) == NULL) {
141
142
                   MPI_Abort(MPI_COMM_WORLD, 1);
143
              fclose(fp);
144
              fp = NULL;
145
146
147
```

```
MPI_Barrier(MPI_COMM_WORLD);
148
          MPI_Bcast(A, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
149
150
           /* Start Timing */
151
          if(rank == 0) {
152
               sec = timer();
153
154
155
156
          for (k = 0; k < N - 1; k++) {
157
158
               block_rows = counts[rank];
               bcaster = get_bcaster(ccounts, bcaster);
159
160
161
               MPI_Barrier(MPI_COMM_WORLD);
               debug(" broadcaster is %d\n", bcaster);
162
               MPI_Barrier(MPI_COMM_WORLD);
163
164
               \label{eq:mpi_bcast} $$ MPI_Bcast(\&A[k * N], N, MPI_DOUBLE, bcaster, MPI_COMM_WORLD); $$
165
166
               process_rows(k, rank, N, max_rank, block_rows, displs, A);
167
168
169
          MPI_Barrier(MPI_COMM_WORLD);
          if(rank == 0) {
170
               sec = timer();
171
               printf("Calc Time: %lf\n", sec);
172
173
174
          ret = MPI_Finalize();
175
          if(ret == 0) {
176
               debug("%d FINALIZED!!! with code: %d\n", rank, ret);
177
178
179
          else {
               debug("%d NOT FINALIZED!!! with code: %d\n", rank, ret);
180
181
182
183
          if(rank == (max_rank - 1)) {
              fp = fopen(argv[2], "w");
184
185
               fprint_matrix_2d(fp, N, N, A);
               fclose(fp);
186
187
188
          free(A);
          free(counts);
189
          free(ccounts);
190
191
          return 0;
192
     }
193
```