src/parallel.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <omp.h>
5
6  /*#define DEBUG 1*/
7  /*#define BUBBLE 1*/
8  #define THREADS 16
9
10 #define ROWS 10000
11 #define COLUMNS 100000
12 #define WORKTAG 1
13 #define DIETAG 2
14 #define CHUNK 8
15
16 int vet[ROWS][COLUMNS];
17
18 void
19 bs (int n, int * vetor)
20 {
21   int c =0, d, troca, trocou =1;
22
23   while ((c < (n-1)) & trocou )
24     {
25       trocou = 0;
26       for (d = 0 ; d < n - c - 1; d++)
27         if (vetor[d] > vetor[d+1])
28           {
29             troca      = vetor[d];
30             vetor[d]   = vetor[d+1];
31             vetor[d+1] = troca;
32             trocou = 1;
33           }
34       c++;
35     }
36
37 #ifdef DEBUG
38   for ( c = 0 ; c < n ; c++ )
39     printf("%03d ", vetor[c]);
40   printf("\n");
41 #endif
42 }
43
44
45 int
46 compare (const void* a, const void* b)
47 {
48   return *((const int*) a) - *((const int*) b);
49 }
50
51 int
52 master (void)
```

```
53 {
54   double t1,t2;
55   t1 = MPI_Wtime();
56
57   int proc_n;
58   int rank;
59
60   MPI_Status status;
61   MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
62
63   //Populate the matrix
64   int i, j, k;
65   for (i = 0; i < ROWS; i++)
66     {
67       k = COLUMNS;
68       for (j = 0; j < COLUMNS; j++)
69         {
70           vet[i][j] = k;
71           k--;
72         }
73     }
74
75   //Seed the slaves
76   int sent = 0;
77   int received = 0;
78   while ( sent < ROWS )
79     {
80       for (rank = 1; rank < proc_n && sent < ROWS; rank++)
81         {
82           MPI_Send(vet[sent], CHUNK * COLUMNS, MPI_INT, rank, WORKTAG,
                MPI_COMM_WORLD);
83           sent += CHUNK;
84         }
85       for (rank = 1; rank < proc_n && received < ROWS; rank++)
86         {
87           MPI_Recv(vet[received], CHUNK * COLUMNS, MPI_INT, rank,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
88           received += CHUNK;
89         }
90     }
91
92   //Kill the slaves
93   i = 1;
94   while (i < proc_n)
95     {
96       MPI_Send(0, 0, MPI_INT, i++, DIETAG, MPI_COMM_WORLD);
97     }
98
99
100  t2 = MPI_Wtime();
101  fprintf(stderr, "Time: %fs\n\n", t2-t1);
102
103  return 0;
104 }
```

```
105
106 int
107 slave (void)
108 {
109   int proc_n;
110   int my_rank;
111
112   MPI_Status status;
113   MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
114   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
115
116   int work[CHUNK][COLUMNS];
117
118   //Receive and work until it dies
119   while (1)
120     {
121       MPI_Recv(work, CHUNK * COLUMNS, MPI_INT, 0, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
122
123       if (status.MPI_TAG == DIETAG)
124         {
125           return 0;
126         }
127
128       int i;
129       omp_set_num_threads(THREADS);
130 #pragma omp parallel for
131       for (i = 0; i < CHUNK; i++)
132         {
133 #ifdef BUBBLE
134           bs(COLUMNS, work[i]);
135 #else
136           qsort(work[i], COLUMNS, sizeof(int), compare);
137 #endif
138         }
139
140       MPI_Send(work, CHUNK * COLUMNS, MPI_INT, 0, 0, MPI_COMM_WORLD);
141     }
142
143   return 1;
144 }
145
146 int
147 main (int argc, char** argv)
148 {
149   int my_rank;
150   int proc_n;
151
152   MPI_Init(&argc , &argv);
153
154   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
155   MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
156
157   if ( my_rank == 0 )
```

```
158       master();
159     else
160       slave();
161
162     MPI_Finalize();
163
164     return 0;
165   }
```