src/sequential.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 1000
#define COLUMNS 100000

int vet[ROWS][COLUMNS];

int
compare (const void* a, const void* b)
{
  return *((const int*) a) - *((const int*) b);
}

int
main (int argc, const char* argv[])
{
  time_t start, stop;

  start = clock();

  int i, j, k = COLUMNS;
  for (i = 0; i < ROWS; i++)
    {
      for (j = 0; j < COLUMNS; j++)
        {
          vet[i][j] = k;
          k--;
        }
      k = COLUMNS;
    }

  for (i = 0; i < ROWS; i++)
    {
      qsort(vet[i], COLUMNS, sizeof(int), compare);
    }

  stop = clock();

  float diff = ((float)(stop - start) / 1000000.F) * 1000;
  printf("Time: %.0fms\n\n", diff);

  return 0;
}
```

src/parallel.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define ROWS 1000
#define COLUMNS 100000
#define WORKTAG 1
#define DIETAG 2

int vet[ROWS][COLUMNS];

int
compare (const void* a, const void* b)
{
  return *((const int*) a) - *((const int*) b);
}

int
master (void)
{
  double t1,t2;
  t1 = MPI_Wtime();

  int proc_n;
  int rank;
  int work = 0;

  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

  //Populate the matrix
  int i, j, k;
  for (i = 0; i < ROWS; i++)
    {
      k = COLUMNS;
      for (j = 0; j < COLUMNS; j++)
        {
          vet[i][j] = k;
          k--;
        }
    }

  //Seed the slaves
  for (rank = 1; rank < proc_n; rank++)
    {
      MPI_Send(vet[work], COLUMNS, MPI_INT, rank, WORKTAG, MPI_COMM_WORLD)
          ;
      work++;
    }

  //Receive a result from any slave and dispatch a new work request
  int save_path = 0;
```

```
52    while (work < ROWS)
53      {
54        MPI_Recv(vet[save_path], COLUMNS, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
55        MPI_Send(vet[work], COLUMNS, MPI_INT, status.MPI_SOURCE, WORKTAG,
              MPI_COMM_WORLD);
56        work++;
57        save_path++;
58      }
59
60    //Receive last results
61    for (rank = 1; rank < proc_n; rank++)
62      {
63        MPI_Recv(vet[save_path], COLUMNS, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
64        save_path++;
65      }
66
67    //Kill all the slaves
68    for (rank = 1; rank < proc_n; rank++)
69      {
70        MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
71      }
72
73    t2 = MPI_Wtime();
74    fprintf(stderr, "Time: %fs\n\n", t2-t1);
75
76    return 0;
77  }
78
79  int
80  slave (void)
81  {
82    int* work = malloc(COLUMNS * sizeof(int));
83    MPI_Status status;
84
85    //Receive and work until it dies
86    while (1)
87      {
88        MPI_Recv(work, COLUMNS, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
              status);
89
90        if (status.MPI_TAG == DIETAG)
91          {
92            free(work);
93            return 0;
94          }
95
96        qsort(work, COLUMNS, sizeof(int), compare);
97
98        MPI_Send(work, COLUMNS, MPI_INT, 0, 0, MPI_COMM_WORLD);
99      }
100
101   return 1;
```

```
102 }
103
104 int
105 main (int argc, char** argv)
106 {
107   int my_rank;
108   int proc_n;
109
110   MPI_Init(&argc , &argv);
111
112   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
113   MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
114
115   if ( my_rank == 0 )
116     master();
117   else
118     slave();
119
120   MPI_Finalize();
121
122   return 0;
123 }
```