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

//#include <windows.h>


#define TRUE 1;
#define FALSE 0;

/* Local functions */

void Print_list(int local_A[], int local_n, int rank);
void Merge_low(int local_A[], int temp_B[], int temp_C[],
         int local_n);
void Merge_high(int local_A[], int temp_B[], int temp_C[],
        int local_n);
int  Compare(const void* a_p, const void* b_p);

/* Functions involving communication */

void Sort(int local_A[], int local_n, int my_rank,
         int p, MPI_Comm comm);
void Odd_even_iter(int local_A[], int temp_B[], int temp_C[],
         int local_n, int phase, int even_partner, int odd_partner,
         int my_rank, int p, MPI_Comm comm);
void Print_local_lists(int local_A[], int local_n,
         int my_rank, int p, MPI_Comm comm);
void Print_global_list(int local_A[], int local_n, int my_rank,
         int p, MPI_Comm comm);
void Read_list(int local_A[], int local_n, int my_rank, int p,
         MPI_Comm comm);



int main(int argc, char* argv[])
{
   int rank, p; // p : nombre de processeurs

//rank : num  du processeur.  fic machine.mpi modifié pour avoir le pc 002 de rang
0
   int * tl; // tableau local
   int * t ; //tableau global
   int n;  // taille tableau à trier lu dans le fichier
   int nl; // taille des tableaux distribués soit n/p
   MPI_Comm co = MPI_COMM_WORLD;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(co, &p);
   MPI_Comm_rank(co, &rank);

    static const char * filename = "data.txt";
    int val, i;
    char line [50];


// le process 0 lit les données dans le fichier de données et les scattered vers
les autres proc
if (rank ==0)
{
    FILE *fic = fopen(filename,"r");
    if (fic!=NULL)
    {
      i=0;
      // ici on calcule le nombre de données dans le fichier, on crée un tableau
de cette taille
```

```
      // on retourne en début de fichier pour lire l'ensemble des donnée
      while (fgets(line, sizeof(line), fic) != NULL) i++;
      n = i-1;printf("n= %d\n",n) ;//nb de données à trier = taille tableau global
      t=(int*) malloc(n * sizeof(int));
      rewind(fic);
      i=0;
      while (fgets(line, sizeof(line), fic) != NULL && i < n)
      {
        // on lit dans la ligne courante l'entier qu'elle contient ;
        // on suppose que le fichier contient une suite d'entiers, un par ligne
        sscanf(line, "%d", &val);
        t[i] = val;
        //printf("i = %d - ti = %d\n",i, t[i]);
        i++;

      }
    fclose(fic);  // printf("je sors");exit (0);
    }
    else
    {
      perror ( filename );
      printf("Problem ouverture fichier");
      exit(1);
    }
}
    nl = n/p ; // vérifier que p|n
    tl = (int*) malloc(nl*sizeof(int));
    MPI_Scatter(t, n, MPI_INT, tl, nl, MPI_INT, 0, co); // à ce stade chaque pc
possède nl=n/p éléments
 // on est dans le proc 0 donc on libère la mémoire sur le tas

if (rank ==0)
{
   free(t);
    t = NULL;
}
  printf("Proc %d > Before Sort\n", rank);
   fflush(stdout);
  Print_local_lists(tl, n, rank, p, co);


   // pour chaque pc y compris le 0, on tri localement les tableaux locaux tl
   Sort(tl, nl, rank, p,co );

  // Print_local_lists(tl, nl, rank, p, co);
   //fflush(stdout);
printf("after sort : \n");

   Print_global_list(tl, nl, rank, p, co);

   free(tl);
   tl=NULL;

   MPI_Finalize();


    return 0;
}


void swap(int* a, int* b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
```

```
}

void bulle(int tab[], int taille)
{
    for (int i=0; i<taille-1; i++)
        for(int j=i+1; j<taille; j++)
            if (tab[i]>tab[j]) swap(tab+i,tab+j);
}

void bulle_opt(int tab[], int taille)
{
    int swapped = TRUE;
    for (int i=0; i<taille-1; i++)
        for(int j=i+1; j<taille; j++)
            if (tab[i]>tab[j])
            {
              swap(tab+i,tab+j);
              swapped = TRUE;
            }
    if (!swapped) exit(0);

}

void print_tab(int tab[], int taille)
{
    printf("debut tableau\n");
    for(int i=0; i<taille; i++)
        printf("%d\n",tab[i]);
    printf("fin tableau\n");
}



/*-------------------------------------------------------------------
 * Function:    Compare
 * Purpose:     Compare 2 ints, return -1, 0, or 1, respectively, when
 *              the first int is less than, equal, or greater than
 *              the second.  Used by qsort.
 */
int Compare(const void* a_p, const void* b_p) {
   int a = *((int*)a_p);
   int b = *((int*)b_p);

   if (a < b)
      return -1;
   else if (a == b)
      return 0;
   else /* a > b */
      return 1;
}  /* Compare */



/*-------------------------------------------------------------------
 * Function:    Sort
 * Purpose:     Sort local list, use odd-even sort to sort
 *              global list.
 * Input args:  local_n, my_rank, p, comm
 * In/out args: local_A
 */
void Sort(int local_A[], int local_n, int my_rank,
          int p, MPI_Comm comm) {
   int phase;
   int *temp_B, *temp_C;
   int even_partner;  /* phase is even or left-looking */
   int odd_partner;   /* phase is odd or right-looking */
```

```
   /* Temporary storage used in merge-split */
   temp_B = (int*) malloc(local_n*sizeof(int));
   temp_C = (int*) malloc(local_n*sizeof(int));

   /* Find partners:  negative rank => do nothing during phase */
   if (my_rank % 2 != 0) {
      even_partner = my_rank - 1;
      odd_partner = my_rank + 1;
      if (odd_partner == p) odd_partner = MPI_PROC_NULL;  // Idle during odd phase
   } else {
      even_partner = my_rank + 1;
      if (even_partner == p) even_partner = MPI_PROC_NULL;  // Idle during even
phase
      odd_partner = my_rank-1;
   }

   /* Sort local list using built-in quick sort */
   qsort(local_A, local_n, sizeof(int), Compare);

#  ifdef DEBUG
   printf("Proc %d > before loop in sort\n", my_rank);
   fflush(stdout);
#  endif

   for (phase = 0; phase < p; phase++)
      Odd_even_iter(local_A, temp_B, temp_C, local_n, phase,
            even_partner, odd_partner, my_rank, p, comm);

   free(temp_B);
   free(temp_C);
}  /* Sort */


/*--------------------------------------------------------------------
 * Function:    Merge_low
 * Purpose:     Merge the smallest local_n elements in my_keys
 *              and recv_keys into temp_keys.  Then copy temp_keys
 *              back into my_keys.
 * In args:     local_n, recv_keys
 * In/out args: my_keys
 * Scratch:     temp_keys
 */
void Merge_low(
      int  my_keys[],     /* in/out    */
      int  recv_keys[],   /* in        */
      int  temp_keys[],   /* scratch   */
      int  local_n        /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   memcpy(my_keys, temp_keys, local_n*sizeof(int));
}  /* Merge_low */

/*--------------------------------------------------------------------
 * Function:    Merge_high
```

```
 * Purpose:     Merge the largest local_n elements in local_A
 *              and temp_B into temp_C.  Then copy temp_C
 *              back into local_A.
 * In args:     local_n, temp_B
 * In/out args: local_A
 * Scratch:     temp_C
 */
void Merge_high(int local_A[], int temp_B[], int temp_C[],
        int local_n) {
   int ai, bi, ci;

   ai = local_n-1;
   bi = local_n-1;
   ci = local_n-1;
   while (ci >= 0) {
      if (local_A[ai] >= temp_B[bi]) {
         temp_C[ci] = local_A[ai];
         ci--; ai--;
      } else {
         temp_C[ci] = temp_B[bi];
         ci--; bi--;
      }
   }

   memcpy(local_A, temp_C, local_n*sizeof(int));
}  /* Merge_high */


/*---------------------------------------------------------------------
 * Only called by process 0
 */
void Print_list(int local_A[], int local_n, int rank) {
   int i;
   printf("%d: ", rank);
   for (i = 0; i < local_n; i++)
      printf("%d ", local_A[i]);
   printf("\n");
}  /* Print_list */




/*---------------------------------------------------------------------
 * Function:    Odd_even_iter
 * Purpose:     One iteration of Odd-even transposition sort
 * In args:     local_n, phase, my_rank, p, comm
 * In/out args: local_A
 * Scratch:     temp_B, temp_C
 */
void Odd_even_iter(int local_A[], int temp_B[], int temp_C[],
        int local_n, int phase, int even_partner, int odd_partner,
        int my_rank, int p, MPI_Comm comm) {
   MPI_Status status;
   p = 0;
   if (phase % 2 == 0) {
      if (even_partner >= 0) {
         MPI_Sendrecv(local_A, local_n, MPI_INT, even_partner, 0,
            temp_B, local_n, MPI_INT, even_partner, 0, comm,
            &status);
         if (my_rank % 2 != 0)
            Merge_high(local_A, temp_B, temp_C, local_n);
         else
            Merge_low(local_A, temp_B, temp_C, local_n);
      }
   } else { /* odd phase */
```

```
      if (odd_partner >= 0) {
         MPI_Sendrecv(local_A, local_n, MPI_INT, odd_partner, 0,
            temp_B, local_n, MPI_INT, odd_partner, 0, comm,
            &status);
         if (my_rank % 2 != 0)
            Merge_low(local_A, temp_B, temp_C, local_n);
         else
            Merge_high(local_A, temp_B, temp_C, local_n);
      }
   }
}  /* Odd_even_iter */


/*---------------------------------------------------------------------
 * Function:   Print_global_list
 * Purpose:    Print the contents of the global list A
 * Input args:
 *    n, the number of elements
 *    A, the list
 * Note:       Purely local, called only by process 0
 */
void Print_global_list(int local_A[], int local_n, int my_rank, int p,
      MPI_Comm comm) {
   int* A;
   int i, n;

   if (my_rank == 0) {
      n = p*local_n;
      A = (int*) malloc(n*sizeof(int));
      MPI_Gather(local_A, local_n, MPI_INT, A, local_n, MPI_INT, 0,
            comm);
      printf("Global list:\n");
      for (i = 0; i < n; i++)
         printf("%d ", A[i]);
      printf("\n\n");
      free(A);
   } else {
      MPI_Gather(local_A, local_n, MPI_INT, A, local_n, MPI_INT, 0,
            comm);
   }

}  /* Print_global_list */



/*---------------------------------------------------------------------
 * Function:   Print_local_lists
 * Purpose:    Print each process' current list contents
 * Input args: all
 * Notes:
 * 1.  Assumes all participating processes are contributing local_n
 *     elements
 */
void Print_local_lists(int local_A[], int local_n,
        int my_rank, int p, MPI_Comm comm) {
   int*       A;
   int        q;
   MPI_Status status;

   if (my_rank == 0) {
      A = (int*) malloc(local_n*sizeof(int));
      Print_list(local_A, local_n, my_rank);
      for (q = 1; q < p; q++) {
         MPI_Recv(A, local_n, MPI_INT, q, 0, comm, &status);
         Print_list(A, local_n, q);
      }
```

```
      free(A);
   } else {
      MPI_Send(local_A, local_n, MPI_INT, 0, 0, comm);
   }
}  /* Print_local_lists */
```