Sign in/up

carbon

Create and share beautiful images of your source code.
Start typing or drop a file into the text area to get started.

Seti | Auto | Tweet | Export

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

#pragma comment(lib,"msmpi.lib")

//Constants
#define stringLength 20
#define resultElements 10
#define maxDatabaseLength 10000000
#define maxQueryLength 1000

//Holds the whole database
char database[maxDatabaseLength][stringLength];
//Holds the node's part of the database
char databasePart[maxDatabaseLength /2][stringLength];
//Holds the queries
char queries[maxQueryLength][stringLength];

//I am using to store the value and index
struct result {
    int value, index;
};

//Functions used
int readFromFile(char* fileName);
int minThree(int x, int y, int z);
int editDist(char* str1, char* str2, int m, int n);
int compare(const void* s1, const void* s2);


int main(void)
{
    //Variables
    int commCount;
    int numberOfThreads;
    int databaseLength = 0;
    int queryLength = 0;
    int increseQueryElements;
    int local_a, local_n;
    int myrank, comm_sz;
    int color, new_rank, new_comm_sz;
    MPI_Comm New_Comm;

    //2-D lists to store the results in
    struct result** results;
    struct result** finalResult;

    //Initializing the MPI,calc the comm_sz, and calc the myrank
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    //Reading the number of communicators and the number of threads per process
    if (myrank == 0)
    {
        printf("Number of processes is:%d\n", comm_sz);
        printf("Enter the number of communicators:");
        fflush(stdout);
        scanf_s("%d", &commCount);
        printf("Enter the number of threads for each process:");
        fflush(stdout);
        scanf_s("%d", &numberOfThreads);
    }

    //broadcasting the number of communicators and the number of threads
    MPI_Bcast(&commCount, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&numberOfThreads, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Calculating the number of processes per communicator
    int local_ncomm = comm_sz / commCount;

    //Exit if the commCount is smaller than the number of proesses
    if (local_ncomm == 0)
    {
        if (myrank == 0)
        {
            printf("Invalid number of communicators :");
            fflush(stdout);
        }
        MPI_Finalize();
        return 0;
    }
    //if the commCount is not divisible by the processes number
    int remindercomm = comm_sz % commCount;
    if (remindercomm)
    {
        commCount += (remindercomm / local_ncomm);
        local_ncomm = comm_sz / commCount;
    }

    //Creting communicators according to the color
    color = myrank / local_ncomm;
    MPI_Comm_split(MPI_COMM_WORLD, color, myrank, &New_Comm);
    //Getting the new_rank
    MPI_Comm_rank(New_Comm, &new_rank);
    //Gettitng the new_comm_sz
    MPI_Comm_size(New_Comm, &new_comm_sz);

    //Reading the Database and the Queries from the files using the main node of the first comm
    if (myrank == 0)
    {
        //reading the database and the queries from the files
        databaseLength = readFromFile("databaseOneThousand.txt");
        queryLength = readFromFile("queries.txt");

        /*make the database length divisible by the number of nodes by filling the remaining elements with
        a value that does not affect the top ten*/
        //this value does not affect the top ten
        char complete[20] = "1111111111111111111\0";
        int reminder = databaseLength % new_comm_sz;
        //calculating the nedded number of elements to make the database length divisible by the number of nodes
        int temp = 0;
        //filling the needed elements
        if (reminder)
        {
             temp = new_comm_sz - reminder;
            for (int i = databaseLength; i < databaseLength + temp; i++)
            {
                strcpy_s(database[i] , sizeof(complete), complete);
            }
            //increasing the length of datababse with the new length
            databaseLength += temp;
        }


        //the same for the queryLength to be divisible by the commCount
        strcpy_s(complete, sizeof("blank\0"), "blank\0");
        int reminderq = queryLength % commCount;
        //calculating the nedded number of elements to make the database length divisible by the number of nodes
         temp =0;
        //filling the needed elements
        if (reminderq)
        {
            temp = commCount - reminderq;

            for (int i = queryLength; i < queryLength + temp; i++)
            {
                strcpy_s(queries[i], sizeof(complete), complete);
            }
            //increasing the length of queries with the new length
            queryLength += temp;
        }
        increseQueryElements = temp;

    }

    //Broadcasting the database and query lengths
    MPI_Bcast(&databaseLength, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&queryLength, 1, MPI_INT, 0, MPI_COMM_WORLD);

    //Initializing the data division variables
    //the length of the node's part of database
    local_n = databaseLength / new_comm_sz;
    //the first index of my database part
    local_a = local_n * new_rank;
    //my comm query's part length
    int local_nq = queryLength / commCount;
    //if the query length is divisible by the numbber of comms
    int reminderq = queryLength % commCount;
    //the first index of my query part
    int local_aq = new_rank * local_nq;
    //calculating the other comm queries number to use it in the sending and recieving
    int otherCommQueriesElements =  local_nq;

    //Sending the Database and half of the Queries to the other comm main node
    //this is the main node of the first comm sends the database and queries part to the other comm
    if (new_rank == 0)
    {
        if (myrank == 0)
        {
            for (int j = 1; j < commCount; j++)
            {
                //sending the whole database
                MPI_Send(database, databaseLength * stringLength, MPI_CHAR, local_ncomm*j, 0, MPI_COMM_WORLD);
                //sending half of the queries
                MPI_Send(queries + local_nq*j, otherCommQueriesElements * stringLength, MPI_CHAR, local_ncomm*j, 0, MPI_COMM_WORLD);
            }

        }
        //this is the main node of the second comm receiving the databse and the queries part from the first comm
        else
        {
            //recieving the whole database
            MPI_Recv(database, databaseLength * stringLength, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
            //recieving half of the queries
            MPI_Recv(queries, local_nq * stringLength, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
        }
    }


    //The main node of every comm Scatters the Database among nodes
    MPI_Scatter(database, local_n * stringLength, MPI_CHAR, databasePart, local_n * stringLength, MPI_CHAR, 0, New_Comm);

    //The main node of every comm Broadcasts the comm part of the queries among all the nodes
    MPI_Bcast(queries, local_nq * stringLength, MPI_CHAR, 0, New_Comm);

    //Allocating memory space to store the results in
    results = malloc(sizeof(struct result*) * local_nq);
    for (int i = 0; i < local_nq; i++)
    {
        results[i] = malloc(sizeof(struct result) * local_n);
    }

    //Every node creates numberOfThreads and starts working on its Queries according to its part of Database using OpenMP
    int i, j;
#   pragma omp parallel num_threads(numberOfThreads) \
     default ( none ) shared ( results,local_n,local_nq ,queries,databasePart,local_a) private ( i , j )
    //Looping over the queries elements
    for (i = 0; i < local_nq; i++)
    {
#       pragma omp for
        //every query element is compared with every element in the node's part of the database using multible thread
        for (j = 0; j < local_n; j++)
        {
            //calaculating the distance
            results[i][j].value = editDist(queries[i], databasePart[j], strlen(queries[i]), strlen(databasePart[j]));
            //storing the index of the element the query element was compared with. the + local_a makes the index matches the database not the databbasePart
            results[i][j].index = j + local_a;
        }
    }

    //Every node sorts its results
    for (int i = 0; i < local_nq; i++)
    {
        //Sorting the results
        qsort(results[i], local_n, sizeof(struct result), compare);
    }

    //Allocating memory space for the finalResult to gather the results from all the nodes in the same comm in
    finalResult = malloc(sizeof(struct result*) * queryLength);
    if (new_rank == 0)
    {
        //Just the main node of every comm will execute this
        for (int i = 0; i < queryLength; i++)
        {
            finalResult[i] = malloc(sizeof(struct result) * resultElements * new_comm_sz);
        }
    }

    //The main node of every comm Gathers the results into finalResult
    for (int i = 0; i < local_nq; i++)
    {
        //Gathering a qury element results one at a time
        MPI_Gather(results[i], min( resultElements,local_n), MPI_DOUBLE, finalResult[i], min(resultElements, local_n), MPI_DOUBLE, 0, New_Comm);
    }

    //The main node of every comm sorts the finalResult it gathered
    if (new_rank == 0)
    {
        for (int i = 0; i < local_nq; i++)
        {
            //sorting every query element one at a time
            qsort(finalResult[i], min(resultElements, local_n) * new_comm_sz, sizeof(struct result), compare);
        }
    }

    //One comm main node sends its finalResult gathered from the other nodes to the main node of the other comm
    if (new_rank == 0)
    {
        if (myrank == 0)
        {
            for (int j = 0; j < commCount-1; j++)
            {
                for (int i = 0; i < otherCommQueriesElements; i++)
                {
                    //I used MPI_DOUBLE since its size is the same as my struct, this makes it easy to send a struct with size 8
                    MPI_Recv(finalResult[i + local_nq+(j*otherCommQueriesElements)], min(resultElements, local_n) * new_comm_sz, MPI_DOUBLE, local_ncomm*(j+1), 0, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
                }
            }

        }
        else
        {
            for (int i = 0; i < local_nq; i++)
            {
                //the main node sends its results to the other comm msin node
                MPI_Send(finalResult[i], min(resultElements, local_n) * new_comm_sz, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            }
        }
    }


    //The main node of the first comm prints the results
    if (myrank == 0)
    {
        for (int i = 0; i < queryLength-increseQueryElements; i++)
        {

                for (int j = 0; j < min(resultElements, local_n); j++)
                {
                    printf("first Letter: %s, Second letter: %s ,Index: %d , Distance: %d\n", queries[i], database[finalResult[i][j].index], finalResult[i][j].index, finalResult[i][j].value);
                    fflush(stdout);

                }
                printf("-------------------------------------------------\n");

        }
    }

    fflush(stdout);
    MPI_Comm_free(&New_Comm);
    MPI_Finalize();

    char exit;
    if (myrank == 0)
    {
        printf("Enter a letter to exit:");
        fflush(stdout);
        scanf_s("%s", &exit, 1);
    }

    return 0;
}
//A function that reads the database and the queries from the files
int readFromFile(char* fileName)
{
    FILE* fptr = NULL;
    int i = 0;
    int err = fopen_s(&fptr, fileName, "r");
    //If it is the database.txt file
    if (fileName[0] == 'd')
    {
        //Copying the content of the file into the database array
        while (err == 0 && fgets(database[i], 10, fptr)) {
            //Removing the \n from the string
            if(database[i][strlen(database[i]) - 1] == '\n')
                database[i][strlen(database[i]) - 1] = '\0';
            i++;
        }
    }
    //If it is the queries.txt file
    else
    {
        //Copying the content of the file into the queries array
        while (err == 0 && fgets(queries[i], 20, fptr)) {
            //Removing the \n from the string
            if (queries[i][strlen(queries[i]) - 1] == '\n')
                queries[i][strlen(queries[i]) - 1] = '\0';
            i++;
        }
    }

    return i;
}
//A function that finds the minimum of three used in the editDist function
int minThree(int x, int y, int z)
{
    return min(min(x, y), z);
}
//the recursion edit distance algorithm
int editDist(char* str1, char* str2, int m, int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0)
        return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0)
        return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m - 1] == str2[n - 1])
        return editDist(str1, str2, m - 1, n - 1);

    // If last characters are not same, consider all three
    // operations on last character of first string,
    // recursively compute minimum cost for all three
    // operations and take minimum of three values.
    return 1
        + minThree(editDist(str1, str2, m, n - 1), // Insert
            editDist(str1, str2, m - 1, n), // Remove
            editDist(str1, str2, m - 1,
                n - 1) // Replace
        );
}
//I am using it in sorting the arrays of struct result acoording to the value
int compare(const void* s1, const void* s2)
{
    struct result* e1 = (struct result*)s1;
    struct result* e2 = (struct result*)s2;
    return e1->value - e2->value;
}
```