Compilazione da linea di comando con OPENMP

Compilazione del file .c

Il file punto c viene compilato con la direttiva :

```
        consoleU@:~$ gcc -c nomefile.c -o nomefile.o -fopenmp

        consoleU@:~$ gcc -o nomefile nomefile.o -fopenmp (librerie di
seguito)

        consoleU@:~$ (time) ./nomefile
```

Direttive OMP

Il pragma supporta diverse tipologie di costrutti che permettono una esecuzione parallela diversa sui thread a disposizione.

1. ***PRAGMA PARALLEL*:**

   la direttiva permette di poter parallelizzare l'esecuzione dell' intero blocco funzionale successivo sui thread messi a disposizione all'esecuzione del programma

```
        #pragma omp parallel [clause[[, ]clause] · · · ]
            structured block or graph
```

Clausole della pragma parallel:

- **if**(espressione o uno scalare);

  ```
  permette di condizioneare l'esecuzione del pragma
  ```

- **num_threads**(intero o espressione);

  ```
  sovrascrive OMP_NUM_THREADS
  ```

- **shared**(lista delle variabili);

```
  specifica quali sono le variabili condivise tra i thread
```

- **private**(lista delle variabili);

```
  specifica quali variabili sono private e accessibili unicamente al
thread che le utilizza
```

- **default**(none|shared|empty);

```
  specifica una politica di gestione per le variabili non specificate
nelle clausole private e shared. Il valore empty imposta di default
private, il valore shared imposta la politica di shared variables e
none blocca l'esecuzione nel caso in cui per almeno una variabile non
sia stata specificata la politica di accesso.
```

Esempio

Ogni thread messo a disposizione per la parallelizzazione del programma esegue il blocco funzionale sottostante, in questo caso il ciclo for.

Otterremo un output dato da **OMP_NUM_THREADS** esecuzioni concorrenti del for.

```
    #pragma omp parallel shared(n) private (i)
    {
        #pragma omp for
            for (i = 0; i<n; ++i)
                printf("Thread %d, executes iteration
%d\n",omp_get_thread_num(), i);
    }
```

2. ***PRAGMA FOR***:

   Il pragma for permette di poter suddividere l'esecuzione di un singolo for sull'insieme di thread a disposizione. Ciò permette di eseguire una sola volta il for, ma aumentare le prestzioni tramite la parallelizzazione.

```
        #pragma omp parallel for[clause[[, ]clause] · · · ]
            structured block or graph
```

Clausole del pragma for

- **private**(lista delle variabili);

```
  specifica quali variabili sono private e accessibili unicamente al
thread che le utilizza
```

- **firstprivate**(lista delle variabili);

```
  specifica che la variabile, esistente già prima della sezione
parallela, abbia modalità di accesso privata e che il suo valore sia
inizializzato allo stesso valore ad essa precedentemente assegnato
```

- **last private** (lista delle variabili);

```
  specifica che al termine della sezione parallela, la variabile
esterna corrispondente assume il valore che la corrispondente
variabile interna assume all'ultima iterazione del for o all'ultima
sezione del pragma
```

- **ordered**;

```
  specifica che il blocco deve essere eseguito in ordine,
indipendentemente dall'esecuzione dei thread.
```

- **nowait**

```
  fa sì che l'esecuzione della sezione parallela corrente, che di
default avviene in seguito al termine della sezione parallela
precedentee, avvenga immediatamente, senza aspettare la BARRIERA
```

- **reduction**(operator: lista di variabili);

```
  specifica che una o più variabili, che sono private per ogni thread,
sono soggetta a una operazione di riduazione alla fine della regione
parallela.
  L'operazione di riduzione consiste nell'applicazione di un operatore
aritmetico-logico sull'insieme dei valori assunti da quella variabile
nelle varie esecuzioni parallele

  + : (v);
  * : (v);
```

```
- : (v);
& : (~ v);
| : (v);
^: (v);
&& : (v);
|| : (v);
```

- **schedule**(kind [, chunk_size])

```
  la clausola controlla la modalità di divisione tra i thread della
computazione da eseguire per quel blocco funzionale
```

Definizione di chunck size

La **chunck size** rappresenta la granularità del carico di lavoro assegnato ad un thread nella threadpool. Deve essere necessariamente un **intero positivo** e costante all'interno dell'esecuzione.

Esistono 4 modalità di esecuzione della schedule (kind):

- **static**:

```
le  iterazioni sono  divise in blocchi (chunck) e assegnate
staticamente ai thread in ROUND ROBIN. Se la grandezza dei chunk
non è specificata, allora lo spazio delle iterazioni viene diviso
equamente.
```

- **dynamic**:

```
le  iterazioni sono  assegnate ai thread in maniera dinamica alla
richiesta del thread, immediatamente dopo il termine
dell'operazione precedente.
Se la dimensione del chunck non è specificata, di default è 1.
```

- **guided**:

```
analogamente al caso dinamico, ma se il chunck è di grandezza 1,
la grandezza del chunck è proporzionale al numero di iterazioni
assegnate diviso il numero di thread. Se chunck size è k allora
la assegnazione è come precedentemente descritto, ma deve
contenere almeno k iterazioni.
```

- **runtime**:

```
le  decisioni di divisione sono prese a runtime. Si può
specificare TIPO DI SCHEDULE e DIMENSIONE DEL CHUNCK attraverso
la variabile di ambiente OMP_SCHEDULE.
```

Esempio

Ogni iterazione del ciclo for verrà assegnata a uno dei thread messi a disposizione per l'esecuzione.

Otterremo un output dato da un'unica esecuzione del ciclo for, ma parallelizzata tra **OMP_NUM_THREADS** thread diversi.

```
    #pragma omp parallel for shared(n) private (i)
        for (i = 0; i<n; ++i)
            printf("Thread %d, executes iteration
%d\n",omp_get_thread_num(), i);
```

3. ***PRAGMA SECTIONS***:

   Il pragma sections permette la divisione della sezione da parallelizzare in sottosezioni ognuna delle quali viene eseguita da un thread differente. Nel caso in cui i thread disponibili sono maggiori rispetto alle sezioni da assegnare, allora i restanti sono in idle.

```
        #pragma omp sections [clause[[, ]clause] · · · ]
        {
            #pragma section {
                structured block or graph
            }
            #pragma section {
                structured block or graph
            }
        }
```

Clausole del pragma sections

- **private**(lista delle variabili);
- **fistrprivate**(lista delle variabili);
- **lastprivate**(lista delle variabili);
- **nowait**;
- **reduction**(operator: lista di variabili);

Esempio

Ogni iterazione del ciclo for verrà assegnata a uno dei thread messi a disposizione per l'esecuzione. La nowait, inoltre ci permette di non rispettare le bariere invertendo, eventualmente, l'ordine dei thread.

Otterremo un output dato da un'unica esecuzione del ciclo for, ma parallelizzata tra **OMP_NUM_THREADS** thread diversi.

```
    #pragma omp parallel
        {

            #pragma omp sections nowait
            {

                #pragma omp section
                    for (int i=0;i<n;++i)
                        printf("I'm always the thread  number %d, at
iteration n",omp_get_thread_num(),i);

                #pragma omp section
                    for (int i=0;i<n;++i)
                        printf("I'm always the thread  number %d, at
iteration %d\n",omp_get_thread_num(),i);

            }

        }
```

4. ***PRAGMA SINGLE***:

   Il pragma single specifica che il blocco di istruzioni ad esso associato viene eseguito una sola volta da un thread della threadpool, non necessariamente il principale.

   L'esecuzione del blocco nel pragma single forza gli altri thread all'attesa della sua esecuzione. Nel caso in cui il **nowait** sia specificato, invece, l'esecuzione di questi ultimi può avvenire.

```
        #pragma omp single [clause[[, ]clause] · · · ]
            structured block
```

Clausole del pragma sections

- **private**(lista delle variabili);
- **fistrprivate**(lista delle variabili);
- **nowait**;

Esempio

All'interno del blocco parallelizzato la pragma single permette **l'esecuzione del blocco funzionale** associato una sola volta, forzando gli altri thread all'idle.

```
#pragma omp parallel
    {

        #pragma omp parallel shared(a,b) private (i)
            {
                #pragma omp single
                {
                    a=42;
                    printf("Single here by %d\n",omp_get_thread_num());
                }

                #pragma omp for
                    for (i=0;i<n;++i)
                        b[i]=a;
                }
                printf("After parallel : \n");
                for (i=0;i<n;++i)
                    printf("b[%d]=%d\n",i,b[i]);

            }

        }
```

5. ***ALTRI COSTRUTTI DEL PRAGMA*:**

- **Barrier;**

```
#pragma omp barrier
```

Definisce una barriera esplicita.

- **Ordered;**

```
#pragma omp ordered structured – block
```

Appare sempre all'interno di un **pragma (parallel) for** e impone che il blocco di istruzioni successivo avvenga secondo l'ordine del for.

- **Critical**;

```
#pragma omp critical [(name)] structured – block
```

Critical permette di specificare che il successivo blocco di istruzioni deve essere eseguito da un **unico thread per volta**. Solitamente viene utilizzato per implementare un meccanismo di mutua esclusione sulle risorse.

- **Atomic**;

```
#pragma omp atomic structured – block
```

Il costrutto atomic assicura che una locazione di memoria specifica venga acceduta atomicamente, cioè **impedisce la lettura o la scrittura simultanea** da parte di più thread, evitando valori indeterminati.

Esecuzione parallela con message passing MPI

Il middelware MPI viene impiegato in casi in cui si vuole implementare una comunicazione tra più unità di calcolo basandosi su una architettura a memoria distribuita.

Le architetture di questo tipo possono avere diverse topologie, date dalla **distribuzione dei nodi** al loro interno. Nonostante la memoria sia distribuita tra i nodi, attraveerso il meccanismo di message passing, viene fornita l'astrazione di una memoria centralizzata. Inoltre, il concetto di nodo prescinde dalla sua natura architetturale in quanto MPI fa' da middelware di supporto alla astrazione del nodo.

Message passing

Il meccanismo alla base di MPI e del concetto di memoria distribuità è la comunicazione implementata mediante **message passing**. I dati nelle memorie locali di ogni nodo vegnono condivisi mediante un meccanismo di comunicazione basasto sullo scambio di messaggi.

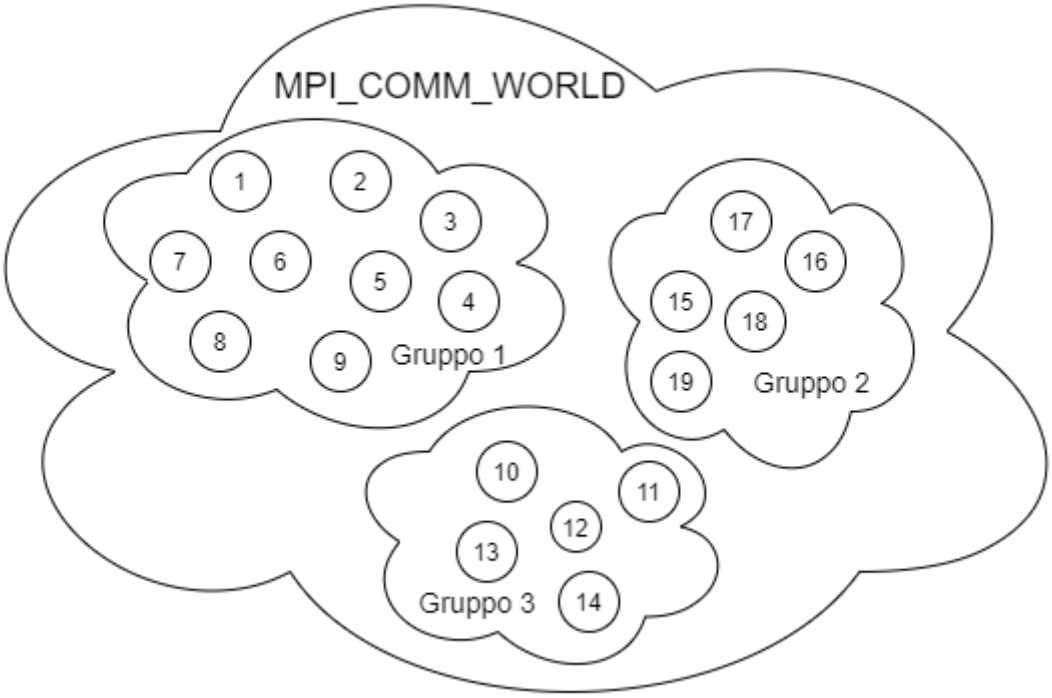
Le metodologie di scambio di messaggi possono essere di diverso genere:

- comunicazioni **sincrone**, mediante le quali si implementa una comunicazione peer-to-peer di tipo **bloccante**.
- comunicazioni **asincrone**, ediante le quali si possono implementare comunicazioni di tipo **non bloccante**.
- comunicazioni **non peer-to-peer**, definite come comunicazioni collettive nelle quali prendono parte tutti i nodi della rete.

Il framework MPI

Il framework MPI è un middelware costituito da più librerie in grado di implementare l'esecuzione parallela su architetture a memoria distribuita.

MPI lavora creando dei processi all'interno dei vari nodi sui quali vengono distribuite le istruzioni. La configurazione ottima si ha quando vengono creati tanti processi quanti sono i core a disposizione nella rete.



MPI crea un insieme di processi che sono racchiusi nell'insieme **MPI_COMM_WORLD**. AL suo interno sono suddivisi in **gruppi**. E'importante notare che i processi di un singolo gruppo non devono necessariamente appartenere allo stesso nodo, prescindendo dalla architettura sottostante.

Si definiscono due concetti fondamentali per MPI:

- **gruppo**: ovvero un insieme logico di processi. Se un gruppo è costituito da p processi, allora verrà assegnato un **rank** a ogni processo, partendo da 0 a p-1.
- **communicator**: Una struttura che permette a un insieme di processi di comunicare tra loro. In ogni communicator deve essere assegnato un rank a ogni processo che va da 0 a p, dove p è il numero di processi. **MPI_COMM_WORLD** è il communicator di default per tutti i processi. La **size** del communicato è data dal numero dei suoi processi e non può essere cambiata dopo la sua creazione.

Compilare un programma MPI

Mediante MPICH è possibile compilare e linkare automaticamente un file .c con le librerie di MPI.

```
consoleU@:~$ mpicc nomefile.c -o nomefile
```

```
consoleU@:~$ mpirun -np numero_processi_da_istanziare ./nomefile
```

Attraverso **numero_processi_da_istanziare** posso specificare quanti processi MPI deve creare per l'esecuzione parallela del programma.

Esempio

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

int main (int argc, char * argv[]){

    int rank, size;
    MPI_Init(&argc, &argv); //init MPI environment
    //get rank and size of communicator
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello! I am rank # %d of %d processes\n", rank,size);

    MPI_Finalize(); //Terminate MPI execution env.
    exit(EXIT_SUCCESS);

}
```

- **MPI_INIT**: inizializza l'environement MPI prendedo gli argomenti a linea di comendo e legge il numero di processi da inizializzare;
- **MPI_COMM_RANK**: prende il rank del processo che sta eseguendo l'istruzione, leggendono da MPI_COMM_WORLD, e lo assegna alla variabile rank.
- **MPI_COMM_SIZE**: legge da MPI_COMM_WORLD il numero dei processi istanziati e assegna il valore alla variabile size.
- **MPI_FINALIZE**: termina l'esecuzione dell'environement MPI.

Comunicazioni MPI: *sincrona bloccante*

Comunicazione peer-to-peer dove sono definiti un mittente e un ricevente.
Il peer ricevente aspetta la **ricezione completa** del messaggioda parte del peer mittente. Il mittente, in seguito all'invio del messaggio, si mette in ascolto dell' **ACK di conferma** da parte del ricevente.

Blocking send

```
    int MPI_Send(void *buf, int count, MPI_Datatype datatype,int dest, int
tag, MPI_Comm comm)
```

Parametri:

- ***buf**: puntatore a **un'area di memoria** dove risiedono i dati da inviare;
- **count**: **numero di elementi** che sono memorizzati nell'area puntata da buf. Esso è un intero non negativo;
- **datatype**: tipologia di dati memorizzati scelti tra gli MPI DATA TYPES.
    - **MPI_CHAR**;
    - **MPI_INT**;
    - **MPI_FLOAT**;
    - **MPI_DOUBLE**;
    - **MPI_LONG**;
    - in aggiunta ci sono anche **strutture** di MPI_DATA_TYPES, **array** indicizzati e **datatypes personalizzati**.
- **dest**: il **rank** del processo destinatario.
- **tag**: un tag che è indicato per identificare il tipo di comunicazione, quindi definire dei **sottocanali** di comunicazione.
- **comm**: specifica il **COMMUNICATOR** doe il processo sta inviando i dati;

Differenza tra MPI_Send e MPI_SSend

La prima termina la sua esecuzione nel momento in cui il buffer di invio è vuoto e quindi riusabile, **senza attendere la corretta ricezione dei dati** da parte del processo ricevente.
La seconda, definita come SECURE SEND, **attende sempre che il ricevente abbia terminato la ricezione dei dati** prima di terminare la propria esecuzione, nonstante il buffer sia vuoto.

Blocking recieve

```
    int MPI_Recv(void *buf, int count, MPI_Datatype datatype,int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

Parametri:
I parametri sono analoghi al caso precedente fatta eccezione per:

- **source**: il quale indica il **rank del processo mittente**;
- **MPI_STATUS**: contenente **informazioni sullo stato del messaggio ricevuto**.

Esempio

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

    int main(int argc, char** argv) {

        int rank, n_ranks;
        int numbers = 5;
        int send_message[numbers];
        int recv_message[numbers];
        MPI_Status status;

        // First call MPI_Init
        MPI_Init(&argc, &argv);

        // Get my rank and the number of ranks
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &n_ranks);

        // Generate numbers to send
        int sign=-1;
        if (rank == 0)
            sign = 1;

        for( int i=0; i<numbers; i++){
            send_message[i] = sign * (i+1);
        }

        if( rank == 0 ){
            // Rank 0 will send first
            MPI_Send(send_message, numbers, MPI_INT, 1, 0, MPI_COMM_WORLD);
        }

        if( rank == 1 ){
            // Rank 1 will receive it's message before sending
            MPI_Recv(recv_message, numbers, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
            printf("Message received by rank %d \n", rank);
            for (int i = 0; i<numbers; i++)
                printf("%d--%d ",rank,recv_message[i]);
            printf("\n");
        }

        if( rank == 1 ){
            // Now rank 1 is free to send
            MPI_Send(send_message, numbers, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }

        if( rank == 0 ){
            // And rank 0 will receive the message
            MPI_Recv(recv_message, numbers, MPI_INT, 1, 0, MPI_COMM_WORLD,
&status);
            printf("Message received by rank %d \n", rank);
            for (int i = 0; i<numbers; i++)
```

```
            printf("%d--%d ",rank,recv_message[i]);
            printf("\n");

        }

        // Call finalize at the end
        return MPI_Finalize();

    }
```

E' necessario implementare una logica di tipo **sender-recieve alternata** in quanto, nel caso in cui volessimo implementare la comunicazione reciprova contemporanea, si incorrerebbe in un errore causato da una **deadlock**.

SendRecieve call

```
    int MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype
senddatatype, int dest, int endtag, void* recvbuf, int recvcount,
MPI_Datatype recvdatatye, int src, int recvtag,MPI_Comm comm, MPI_Status *
status);
```

Parametri:

- **sendbuf**: specifica il **buffer del processo mittente**;
- **sendcount**: specifica il **numero degli elementi del send buffer**;
- **senddatatype**: specifica il **tipo di dato** inviato dal mittente (mpi_data_types precedenti);
- **dst**: specifica il **rank del processo ricevente**;
- **sendtag**: tag del messaggio inviato;
- **recvbuffer**: specifica il **buffer del processo ricevente**;
- **recvcount**: specifica il **numero di elementi nel recv buffer**;
- **recvdatatype**: specifica il **tipo di dato** ricevuto (mpi_data_types precedenti);
- **src**: specifica il **rank del processo mittente**;
- **recvtag**: tag del messaggio ricevuto;
- **comm**: specifica il **COMMUNICATOR** in cui sta avvenendo la comunicazione;
- **status**: struttura dati contenente **informazioni sullo stato del messaggio ricevuto**.

Esempio

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

    #define numbers 3
```

```
    int main(){
        int rank,size;
        MPI_Init(NULL,NULL);
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Status status;

        int *sendbuf,*recvbuf;
        if ( NULL == (sendbuf = malloc( 10 * sizeof (int) ))) exit(1);
        if ( NULL == (recvbuf = malloc( 10 * sizeof (int) ))) exit(1);

        for ( int i = 0 ; i < numbers; i++) {
            sendbuf[i] = rank;
            recvbuf[i] = -100;
        }

        int next_rank, prev_rank;
        next_rank = (rank < size -1) ? rank + 1 : MPI_PROC_NULL;
        prev_rank = (rank >    0   ) ? rank - 1 : MPI_PROC_NULL;

        for ( int i = 0 ; i < numbers; i++)
            printf("[Before] Send Rank: %d\tRecv Rank:
%d\n",sendbuf[i],recvbuf[i]);

        MPI_Sendrecv( sendbuf, numbers, MPI_INT, next_rank, 0, recvbuf,
numbers, MPI_INT, prev_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

        for ( int i = 0 ; i < numbers; i++)
            printf("[After] Send Rank: %d\tRecv Rank:
%d\n",sendbuf[i],recvbuf[i]);


        free(sendbuf);
        free(recvbuf);

        MPI_Finalize();
        return 0;
    }
```

Comunicazioni MPI: *asincrona non bloccante*

Comunicazione peer-to-peer dove sono definiti un mittente e un ricevente.
Il peer mittente inserisce il messaggio in una **coda di invio** (buffer). Il messaggio viene trasferito dal buffer del peer mittente a quello del ricevente mediante il **middelware MPI**.
Il peer ricevente **all'occorrenza legge il messaggio** e analogamente risponde attraverso l'inserimento del messaggio nel proprio **buffer di risposta**.

Non blocking *send*

```
    int MPI_Isend(void* data, int count, MPI_Datatype datatype, int
destination, int tag, MPI_Comm communicator, MPI_Request* request)
```

Parametri:

- **data**: specifica quale è l'area di memoria dove sono presenti i dati da inviare;
- **count**: specifica il numero di elementi da inviare;
- **datatype**: specifica il tipo di dato da inviare;
- **destination**: specifica il rank del processo destinaario;
- **tag**: tag del messaggio;
- **communicator**: il communicator utilizzato;
- **request**: gestore di richiesta non bloccante (usata per wait e test), viene usata per sapere quando la gestione di una operazione non bloccante è completa;

Non blocking *recieve*

```
    int MPI_Irecv(void* data,int count, MPI_Datatype datatype, int src, int
tag, MPI_Comm communicator, MPI_Request* request);
```

Parametri:

- **data**: specifica quale è l'area di memoria del buffer di ricezione;
- **count**: specifica il numero di elementi nel buffer di ricezione;
- **datatype**: specifica il tipo di dato da ricevere;
- **src**: specifica il rank del processo mittente;
- **tag**: tag del messaggio;
- **communicator**: il communicator utilizzato;
- **request**: gestore di richiesta non bloccante (usata per wait e test), viene usata per sapere quando la gestione di una operazione non bloccante è completa;

Wait

La funzione di wait permette di implementare la sincronizzazione tra i processi che comunicano in modo asicrono. In particolare viene utilizzata in combinazione con la recieve per **implementare l'attesa del termine del trasferimento dei dati al ricevente**.

Il parametro **request** è il medesimo inizializzato nella recieve di riferimento.

```

```

```
int MPI_Wait(MPI_Request* request, MPI_Status* status)
```

Parametri:

- **request**: specifica quale è l'operazione send o recieve da dover sincronizzare;
- **status**: struttura che contiene l'esito della request attesa;

Esempio

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

int main(int argc, char** argv) {
    int rank, n_ranks, neighbour;
    int n_numbers = 5;
    int *send_message;
    int *recv_message;
    MPI_Status status;
    MPI_Request request;
    int return_value;

    send_message = malloc(n_numbers*sizeof(int));
    recv_message = malloc(n_numbers*sizeof(int));

    // First call MPI_Init
    MPI_Init(&argc, &argv);

    // Get my rank and the number of ranks
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_ranks);

    // Call the other rank the neighbour
    if( rank == 0 ){
        neighbour = 1;
    } else {
        neighbour = 0;
    }

    // Generate numbers to send
    for( int i=0; i<n_numbers; i++){
        send_message[i] = i + (rank*10);
    }

    // Send the message to other rank
    MPI_Isend(send_message, n_numbers, MPI_INT, neighbour, 0,
MPI_COMM_WORLD, &request);
```

```
        // Receive the message from the other rank
        MPI_Irecv(recv_message, n_numbers, MPI_INT, neighbour, 0,
MPI_COMM_WORLD, &request);
        MPI_Wait( &request, &status );
        printf("Message received by rank %d \n", rank);

        // Call finalize before freeing messages
        return_value = MPI_Finalize();

        free(send_message);
        free(recv_message);
        return return_value;
    }
```
