David Morillo Massagué (1666540)
Adrià Muro Gómez (1665191)

Pràctica de Laboratori 3 - MPI

Plantejament del problema

El problema que s'estudia en aquesta pràctica és el de l'algoritme 2D de Laplace, el qual té moltes aplicacions (ja vist en format d'una sola dimensió en l'assignatura d'Enginyeria de Rendiment). A part de les operacions matemàtiques que empra aquest, ens interessa saber que:

- Es modela una malla de N files i M columnes, que la denominarem com a matriu A.
- Aquest algoritme consta de X iteracions, en cada una de les quals es fa un "laplace step". Aquest pas aplica els càlculs esmentats en la presentació de la pràctica, per a cada element de la matriu A.
- Les iteracions tenen dependències entre si. És a dir, que no es pot iniciar un "laplace step" abans que hagi acabat el càlcul de l'anterior.
- El funcionament d'aquestes dependències és el següent:

    Per al càlcul del nou valor de l'element A(i, j) necessitem el valor de l'iteració anterior de cada un dels seus veïns. És a dir, que hi ha una espècie de comunicació (per exemple, propagació de calor) entre veïns que avança una posició (en 4 direccions) per cada iteració. Això vol dir que, si separem la matriu (per a fer una paral·lelització) i fem els càlculs per X iteracions per separat, no existirà aquesta transmissió de valors entre elements veïns i el resultats no seran els correctes. És per això que si definim una estratègia de paral·lelització, necessitarem que existeixi una comunicació entre nodes (parts de la matriu) en cada una de les iteracions

Pensem que és possible una paral·lelització eficient fent servir les eines de MPI malgrat aquestes limitacions.

Anàlisi previ del codi

Aquest codi realitza una relaxació de Jacobi en una malla 2D per resoldre l'equació de Laplace. A continuació, analitzarem el codi per identificar les oportunitats de paral·lelització en un entorn multi-node.

Funció *stencil*

```
float stencil(float v1, float v2, float v3, float v4) {
   return (v1 + v2 + v3 + v4) / 4;
}
```

Aquesta funció calcula el valor mig dels quatre valors d'entrada. És un càlcul senzill i totalment paral·lelitzable, ja que cada càlcul és independent dels altres. Però com que això suposa enviar dades (NxM cops per iteració) constantment entre nodes, decidim no implementar canvis en aquesta funció.

Funció *laplace_step*

```
void laplace_step(float *in, float *out, int n, int m) {
   int i, j;
   for (i = 1; i < n - 1; i++)
      for (j = 1; j < m - 1; j++)
         out[i * m + j] = stencil(in[i * m + j + 1], in[i * m + j - 1], in[(i - 1) * m + j], in[(i + 1) *
m + j]);
}
```

Aquesta funció aplica el càlcul del stencil a cada punt de la malla (excepte les vores). Cada punt es calcula de manera independent, per tant, aquesta funció és un candidat ideal per a la paral·lelització. En un entorn multi-node, podem dividir la malla en submalles i assignar cada submalla a un node diferent.

Funció *laplace_error*

```
float laplace_error(float *old, float *new, int n, int m) {
   int i, j;
   float error = 0.0f;
   for (i = 1; i < n - 1; i++)
      for (j = 1; j < m - 1; j++)
         error = fmaxf(error, sqrtf(fabsf(old[i * m + j] - new[i * m + j])));
   return error;
}
```

Aquesta funció calcula l'error màxim entre les dues matrius. Similarment a la funció anterior, cada càlcul de l'error per cada punt és independent, per la qual cosa també es pot paral·lelitzar fàcilment.

Funció *laplace_copy*

```
void laplace_copy(float *in, float *out, int n, int m) {
   int i, j;
   for (i = 1; i < n - 1; i++)
      for (j = 1; j < m - 1; j++)
         out[i * m + j] = in[i * m + j];
}
```

Aquesta funció copia els valors de la matriu “in” a la matriu “out”. Aquesta operació és trivialment paral·lelitzable amb eines com OpenMP o OpenACC, ja que comparteixen memòria, però considerem de poca utilitat amb MPI.

Bucle principal (funció *main*)

```
// Main loop: iterate until error <= tol a maximum of iter_max iterations
while (error > tol && iter < iter_max) {
    // Compute new values using main matrix and writing into auxiliary matrix
    laplace_step(A, Anew, n, m);

    // Compute error = maximum of the square root of the absolute differences
    error = laplace_error(A, Anew, n, m);

    // Copy from auxiliary matrix to main matrix
    laplace_copy(Anew, A, n, m);

    // if number of iterations is multiple of 10 then print error on the screen
    iter++;
    if (iter % (iter_max / 10) == 0)
        printf("%5d, %0.6f\n", iter, error);
}
```

La funció principal configura la malla, i executa un bucle principal fins que l'error és menor o igual a la tolerància especificada o s'assoleix el nombre màxim d'iteracions. Cada iteració consta dels passos següents:

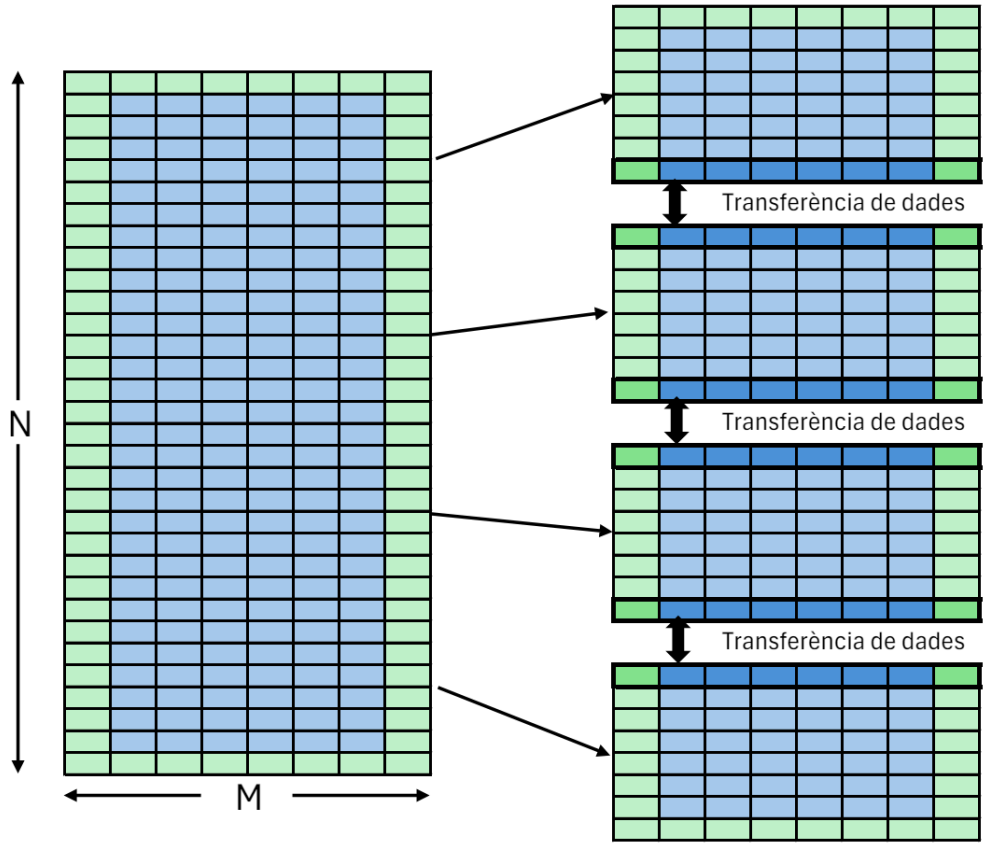
1. Calcula els nous valors (funció **laplace_step**)
    a. Aquesta funció fa servir la funció **stencil**
2. Calcula l'error (funció **laplace_error**)
3. Copia els valors nous a la matriu principal (funció **laplace_copy**)

Estratègia de millora

Com s'ha comentat en l'anàlisi de les funcions una opció de paral·lelització, sense tenir memòria compartida, és la de dedicar el càlcul d'una part de la malla per a cada node del sistema. Com hem remarcat en el primer apartat, també haurien d'existir una sèrie de comunicacions entre nodes per tal de garantir que els valors de la malla es dissipen correctament.

Una observació que cal destacar és que, per tal de donar pas a una altra iteració, només haurem d'enviar els valors de les "fronteres" entre aquestes sub-malles.

En la següent imatge s'il·lustra la idea per a la segmentació de la matriu A per nodes (4 en aquest cas) i la transferència de valors que es faria en cada iteració. Com podem veure, cada node (excepte els dels extrems) hauria d'enviar i rebre dos files de la matriu: la primera i la última (corresponent a cada node).



En la imatge, els elements en verd són les vores, els valors de les quals es tindrà en compte per al càlcul però no es modificarà el seu valor. Els element blaus són els que, per a cada iteració, el seu valor canviarà. El valor d'aquest és evident que també és necessari per al correcte funcionament de la funció *laplace_step*. És per això que si aïllem el còmput de la matriu entre nodes diferents (part esquerra de la imatge), haurem de compartir el valor de les cel·les ressaltades. Això vol dir que enviarem i rebrem X*2*M elements per iteració, on X és el nombre de nodes, i M el nombre d'elements per fila.

*Per a la pràctica hem decidit separar el càlcul de la matriu en files per conveniència, encara que es podria haver fet per columnes.

Aplicació de la paral·lelització

Canvis en les funcions

Per a afavorir la segmentació de càlculs en la matriu A s'han modificat les següents funcions:

```
void laplace_step(float *in, float *out, int start_row, int end_row, int m) {
    int i, j;
    for (i = start_row; i < end_row; i++)
        for (j = 1; j < m - 1; j++)
            out[i * m + j] = stencil(in[i * m + j + 1], in[i * m + j - 1], in[(i - 1) * m + j], in[(i + 1) *
m + j]);
}

float laplace_error(float *old, float *new, int start_row, int end_row, int m) {
    int i, j;
    float error = 0.0f;
    for (i = start_row; i < end_row; i++)
        for (j = 1; j < m - 1; j++)
            error = fmaxf(error, sqrtf(fabsf(old[i * m + j] - new[i * m + j])));
    return error;
}

void laplace_copy(float *in, float *out, int start_row, int end_row, int m) {
    int i, j;
    for (i = start_row; i < end_row; i++)
        for (j = 1; j < m - 1; j++)
            out[i * m + j] = in[i * m + j];
}
```

Hem aplicat uns canvis en el rang del bucles *for*, per a poder computar només una part de la matriu A, respectiu a un node. Això afavoreix la paral·lelització que comentarem a continuació.

Un cop inicialitzada la regió paral·lela i les variables *rank* i *world_size*, en la funció *main*, inicialitzem les matrius A i Anew. Cal destacar que, encara que estigui inicialitzada la matriu completa per cada node, no es realitzarà càlculs en la totalitat d'aquesta, sinó que només es modificarà una part d'aquesta, respectivament al node que li pertany.

Variables auxiliars per node

A partir les les variables *rank* i *world_size*, creem les variables auxiliars:

```
int rows_per_node = n / world_size;
int start_row = rank * rows_per_node + 1;
int end_row = (rank + 1) * rows_per_node;
float local_error;
```

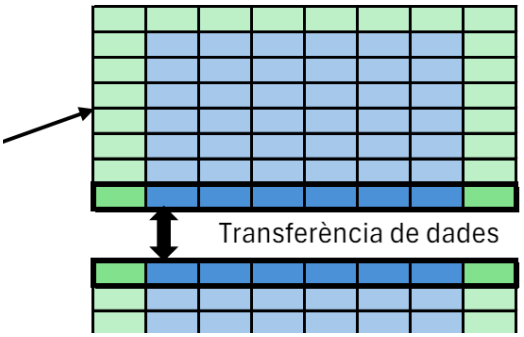
Aquestes ens ajudaran a determinar quin és el tall que hem de fer servir. A cada node se li assigna una variable *start_row* i *end_row* (primera i última fila que modificarà, respectivament). A més, es crea la variable *local_error*, que guardarà el resultat de la funció *laplace_error* per cada node.

Comunicacions

Dins el bucle (while) principal, hem construit les clàusules necessàries per a la comunicació **punt-a-punt** entre files requerida per a la paral·lelització. El tipus que comunicació que s'ha fet servir és la **bloquejant**, ja que s'ha de garantitzar que tots els nodes reben la informació necessària i la actualitzin a la matriu abans de fer un altre pas a la iteració:

```
// Send top and bottom rows with neighboring processes
if (rank > 0) {
    MPI_Send(&A[start_row * m], m, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&A[(start_row - 1) * m], m, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
if (rank < world_size - 1) {
    MPI_Send(&A[(end_row - 1) * m], m, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&A[end_row * m], m, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
```

Com ja hem comentat en la estratègia, només necessitarem enviar i rebre les primeres i últimes files associades a cada node per cada iteració. La comprovació que es fa en els ifs és per a estalviar-nos enviaments de dades innecessaris a cada iteració. El primer comprova si el node actual (rank) és el corresponent a la part superior de la matriu. Si es compleix, no cal que aquest enviï els seus valors, ja que no té elements veïns a dalt (veure imatge de referència). El mateix passa amb la segona comprovació, que estalvia enviar una part de les dades si detecta que es tracta del node amb rank més alt.



En aquesta imatge referent a l'apartat anterior, s'il·lustra les dades que modifica i comparteix el node amb rank=0, corresponent a la primera secció de la matriu A.

Com que només té un node veí, compartirà les dades únicament amb aquest.

A continuació s'expliquen els arguments de les funcions *MPI_Send* i *MPI_Recv* (només en el cas de la comunicació entre el node actual i el seu veí amb rang inferior, per una major simplificació):

```
MPI_Send(&A[start_row * m], m, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD);
```

1. **&A[start_row * m]**: És el punt de memòria des d'on comença a enviar les dades. En aquest cas, és l'adreça de memòria de l'element a la fila *start_row* de la matriu A.
2. **m**: El nombre d'elements que s'envien. Aquí es tracta del nombre d'elements de tipus MPI_FLOAT a enviar.
3. **MPI_FLOAT**: El tipus de dades dels elements a enviar. En aquest cas, cada element és un flotant (float).
4. **rank - 1**: El rang del procés destí. Aquí s'està enviant al procés amb rang immediatament inferior (*rank* - 1).
5. **0**: L'etiqueta del missatge, que és un identificador que permet als processos distingir entre diferents missatges.
6. **MPI_COMM_WORLD**: El comunicador que especifica el conjunt de processos que participen en la comunicació. MPI_COMM_WORLD inclou tots els processos inicialitzats per MPI.

```
MPI_Recv(&A[(start_row - 1) * m], m, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

1. **&A[(start_row - 1) * m]**: És el punt de memòria on es col·locaran les dades rebudes. En aquest cas, és l'adreça de memòria de l'element a la fila *start_row* - 1 de la matriu A.
2. **m**: El nombre d'elements que es reben. Aquí es tracta del nombre d'elements de tipus MPI_FLOAT a rebre.
3. **MPI_FLOAT**: El tipus de dades dels elements a rebre. En aquest cas, cada element és un float.
4. **rank - 1**: El rang del procés emissor. Aquí s'està rebent del procés amb rang immediatament inferior (*rank* - 1).
5. **0**: L'etiqueta del missatge, que és un identificador que permet als processos distingir entre diferents missatges.

6. **MPI_COMM_WORLD**: El comunicador que especifica el conjunt de processos que participen en la comunicació. MPI_COMM_WORLD inclou tots els processos inicialitzats per MPI.

7. **MPI_STATUS_IGNORE**: Paràmetre que indica a MPI que no es necessita informació addicional sobre l'estat de la recepció. Això es fa servir si no és necessari conèixer detalls com el nombre d'elements realment rebuts o la font del missatge.

S'ha decidit construir una comunicació **punt-a-punt** (en comptes de scatter i gather) ja que resulta menys costós que segmentar la matriu A en X nodes per a cada iteració, cosa que no resultaria d'utilitat, ja que, com hem dit, només necessitem compartir els valors de files individuals.

Còmput per node

El codi del bucle contunua de la següent manera:

```
// Compute new values using main matrix and writing into auxiliary matrix
laplace_step(A, Anew, start_row, end_row, m);

// Compute error = maximum of the square root of the absolute differences
local_error = laplace_error(A, Anew, start_row, end_row, m);

// Copy from auxiliary matrix to main matrix
laplace_copy(Anew, A, start_row, end_row, m);
```

La única modificació que s'ha fet és la d'incloure les files d'inici i de fi com a arguments de les funcions prèviament modificades. Cal destacar que la variable *local_error* guarda el valor màxim d'error de cada secció (node). Per a saber l'error màxim de la matriu sencera apliquem la següent reducció:

Reducció

```
// Reduce the error of all nodes
MPI_Allreduce(&local_error, &error, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
```

1. **&local_error**: És el punt de memòria que conté el valor local que es vol reduir. En aquest cas, *local_error* és la variable local que conté l'error calculat pel procés actual.

2. **&error**: És el punt de memòria on es col·locarà el resultat de l'operació de reducció. En aquest cas, *error* és la variable on es guardarà el valor reduït resultant de combinar els valors *local_error* de tots els processos.

3. **1**: El nombre d'elements a reduir. Aquí es tracta d'un sol element de tipus float.

4. **MPI_FLOAT**: El tipus de dades dels elements a reduir. En aquest cas, cada element és un flotant (float).

5. **MPI_MAX**: L'operació de reducció a aplicar. Aquí s'utilitza MPI_MAX per obtenir el màxim valor dels errors locals (*local_error*) de tots els processos.

6. **MPI_COMM_WORLD**: El comunicador que especifica el conjunt de processos que participen en la comunicació. MPI_COMM_WORLD inclou tots els processos inicialitzats per MPI.

Així, la instrucció *MPI_Allreduce* combina els valors *local_error* de tots els processos utilitzant l'operació MPI_MAX per determinar el valor màxim i guarda el resultat en la variable *error.* Gràcies a això, podem comprovar la condició per a una nova iteració del bucle *while*, i imprimir aquests valors per pantalla.

Càlcul del temps de còmput

Per a calcular i mostrar el temps que dedica el programa al còmput del algoritme, inicialitzem dues variables fent servir funcions específiques de MPI i mostrem el temps de còmput:

```
  // Start timer
  start_time = MPI_Wtime();

// ——------------------——---------- BUCLE PRINICPAL ——------------------——----------

  // End timer
  end_time = MPI_Wtime();
  // Print the elapsed time
  if (rank == 0) {
     printf("Total computation time: %f seconds\n", end_time - start_time);
  }
```

Per a fer una deguda comparació de speedups, hem aplicat aquest cronòmetre també en la versió seqüencial:

```
  // Start timer
  clock_t start_time = clock();

// ——------------------——---------- BUCLE PRINICPAL ——------------------——----------

  // End timer
  clock_t end_time = clock();

  // Calculate and print the elapsed time
  double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
  printf("Total computation time: %f seconds\n", elapsed_time);
```

Resultats obtinguts

Proves fetes amb una malla de 4096 x 4096 i 100 iteracions

Execució:	Time elapsed (s)	Speedup	Eficiència
Seqüencial	11.52	1.00	1.00
Par: 1 node	12.11	0.95	0.95
Par: 2 nodes	7.08	1.63	0.81
Par: 4 nodes	3.70	3.11	0.78
Par: 8 nodes	2.77	4.16	0.52
Par: 16 nodes	3.74	3.08	0.19
Par: 32 nodes	4.57	2.52	0.08

Eficiència del Treball de Paral·lelització:

L'eficiència de la paral·lelització disminueix a mesura que s'incrementa el nombre de nodes, especialment a partir dels 8 nodes. Això suggereix que el treball de paral·lelització està limitat per la sobrecàrrega de **comunicació** entre els nodes. En configuracions amb pocs nodes, la paral·lelització és efectiva i proporciona una bona acceleració, però en augmentar el nombre de nodes, els beneficis es veuen contrarestats per la sobrecàrrega.

Possibles Millores

1. **Balanç de Càrrega:** Assegurar-se que la càrrega de treball estigui equilibrada entre tots els nodes per evitar que alguns nodes acabin abans que altres i esperin per sincronitzar-se.

Possibilitats d'Ampliació

1. **Augment de la Mida de la Malla:** Incrementar la mida de la malla (n i m) pot fer que la paral·lelització sigui més eficient, ja que la càrrega computacional seria major en comparació amb la sobrecàrrega de comunicació.
2. **Ús de Més Nodes:** Amb els ajustos adequats en la comunicació i el balanç de càrrega, podria ser beneficiós utilitzar més nodes, especialment si es combinen amb una mida de malla major i més iteracions.

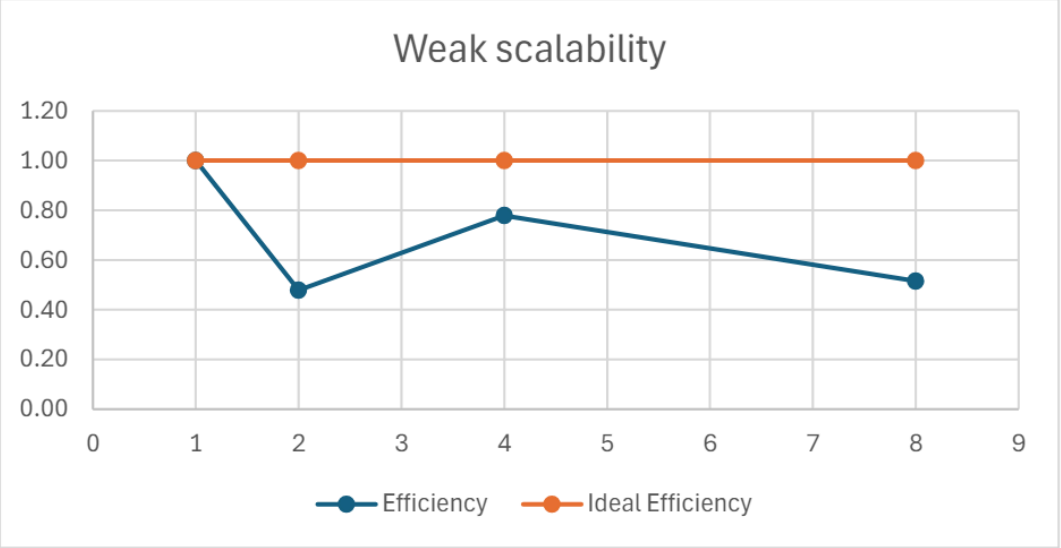
En resum, tot i que la paral·lelització inicial mostra beneficis clars, especialment amb un nombre menor de nodes, l'eficiència disminueix amb l'augment dels mateixos. Amb ajustos i optimitzacions addicionals, és possible millorar l'eficiència i escalar el problema a configuracions més grans.

Escalabilitat en profunditat

Hem elaborat també aquest gràfic on podem apreciar la escalabilitat que té el problema paral·lelitzat, a mesura que augmenta de la mateixa manera, la mida de la matriu, i el nombre de nodes utilitzats:

Weak Scalability (4096 columns)

Rows	N. nodes	Time	Efficiency	Ideal Efficiency
4096	1	11.61	1.00	1
8192	2	24.24	0.48	1
16384	4	14.90	0.78	1
32768	8	22.51	0.52	1



Podem apreciar irregularitats en l'eficiència resultant, degut probablement en factors externs com l'estat del servidor (amb molta demanda al moment d'executar). Aquesta demanda pot afectar a la comunicació entre nodes i al rendiment de cada màquina individual. Per exemple, veiem com amb dos nodes (i el doble de files), l'eficiència no arriba al 50%, però amb el doble de nodes i mida, augmenta gairebé al 80%.