

# Projecte de Laboratori de Pràctiques

## Plantejament problema

El plantejament del problema és el càlcul l'equació de Laplace utilitzant el mètode iteratiu de Jacobi. El problema que ens trobem és que és un càlcul molt gran per a fer-ho amb un sol node. La solució plantejada consisteix en fer un codi paral·lel que faci ús de múltiples nodes per tal de resoldre el problema en un temps raonable. Per paral·lelitzar el problema de Laplace utilitzarem MPI com a mecanisme de comunicació i sincronització.

## Anàlisi del problema

Un cop analitzat el problema, s'ha decidit utilitzar N processos per a la resolució distribuïda del problema. Cadascun d'aquests processos ha de realitzar una part del càlcul sobre una porció de les dades. Això implica tenir en compte els intercanvis necessaris de dades entre els processos i assegurar-ne la sincronització. La distribució de les dades entre els processos determina quins processos han de comunicar-se entre ells.

Per a realitzar aquesta paral·lelització, s'ha modificat el codi original per fer-lo compatible amb l'entorn MPI. Els càlculs que realitzaran els processos es reparteixen de manera que cadascun d'ells s'encarregui d'un tros específic de la matriu, definit per les variables `n_files`, `first_row` i `final_row`. Cada procés rep un rang específic de files per processar, que depèn del seu identificador de procés (`rank`) i del nombre total de processos (`world_size`). Així, ens assurem que cada procés té accés només a les dades que li corresponen.

A més, s'han fet operacions d'enviament i recepció (`MPI_Send` i `MPI_Recv`) per a intercanviar les files necessàries de la matriu entre els processos. En cada iteració, els processos intercanvien les files adjacents de les seves regions de la matriu corresponents amb els processos adjacents. D'aquesta manera, cada procés té accés a les dades actualitzades necessàries per realitzar els càlculs correctament. Així, ens assurem de que es sincronitzin.

És important recalcar que s'ha aplicat una condició per a controlar com es realitzen els càlculs per tenir en compte el nombre de files adjacents de les regions corresponents de la matriu per a cada procés determinat. Per exemple, si un procés processa la primera o la

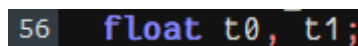
darrera fila, només necessita una fila addicional de la matriu, mentre que la resta de processos necessiten dues files addicionals. Això es deu a la dependència dels càlculs en les vores de la matriu per a la convergència correcta de l'algoritme.

Finalment, s'ha afegit la funció `MPI_Allreduce` per calcular el màxim error de tots els processos i garantir que tots els processos convergeixin al mateix temps. Això és necessari per assegurar que la condició de parada del mètode iteratiu es compleixi correctament en els múltiples nodes.

## Disseny de la solució

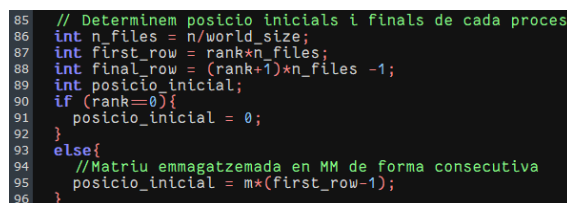
Per tal d'implementar la solució proposada, hem modificat el codi `laplace.c`. La nova implementació l'hem emmagatzemat en l'arxiu `laplace_mpi.c`.

En aquest fitxer, no hem modificat les funcions, només el main. Dins d'aquest, hem afegit les inicialitzacions que es mostren en la Figura 1 i 2. Les variables `t0` i `t1` són per mesurar els temps d'execució. A més, podem observar que hem creat la variable `n_files` per emmagatzemar el número de files que calcularà cada procés. Cal destacar que aquest valor sempre serà múltiple del número de processos. D'altra banda, hem calculat la posició inicial de les files de cada procés. Per obtenir la posició de la última fila, accedim a la posició inicial del procés següent i li restem un. Més endavant, veurem que la variable `posicio_incial` ens servirà per indicar en quina posició comencem a fer els càlculs. En cas que executi el codi el procés 0, aquesta serà 0. En la resta de casos, accedim a la posició de la fila i multipliquem el valor pel número de columnes. El motiu és perquè les posicions de la matriu estan reservades en memòria principal degut al `malloc()`. Altres inicialitzacions realitzades són les variables `error_global` per guardar el valor màxim d'error entre tots els processos. Per últim, utilitzem "`MPI_Status Status`" per revisar l'estat de la recepció dels missatges.



```
56 float t0, t1;
```

Figura 1.



```
85 // Determinem posicio inicials i finals de cada proces
86 int n_files = n/world_size;
87 int first_row = rank*n_files;
88 int final_row = (rank+1)*n_files -1;
89 int posicio_inicial;
90 if (rank==0){
91     posicio_inicial = 0;
92 }
93 else{
94     //Matriu emmagatzemada en MM de forma consecutiva
95     posicio_inicial = m*(first_row-1);
96 }
```

Figura 2.

En segon lloc, hem inicialitzat la variable `t0` amb la funció `MPI_Wtime()`. Tal i com es mostra en la Figura 3, aquesta representa el temps inicial, és a dir, en quin instant de temps comencem a executar el bucle `while`. No tenim en compte les inicialitzacions de les variables i les reserves de memòria ja que és un procés lent i constant. Per tant, el seu valor és menyspreable. Cal destacar que només calcula el temps un procés, en el nostre cas, el procés 0 (tot i que ho podria fer qualsevol altre).

```
101  if (rank==0){
102      t0=MPI_Wtime();
103  }
```

Figura 3.

A continuació, vam programar l'enviament i rebuda de missatges. Recordem que hem de respectar les bores de la matriu. Per tant, tots els processos menys el primer, enviaran la seva primera fila al procés anterior. En conseqüència, rebran la última fila del procés anterior. Excloïm el primer procés ja que aquest no ha de calcular la primera fila de la matriu (ja que és una bora) i, per tant, tampoc no té cap procés anterior que li envii dades. D'altra banda, tots els processos menys l'últim, enviaran l'última fila al següent procés. Així doncs, rebran la primera fila del següent procés. No tenim en compte l'últim procés ja que aquest no ha de calcular la última fila de la matriu (ja que és una bora) i, per tant, tampoc no té cap procés posterior que li envii dades. El codi resultant es mostra en la Figura 4. Podriem haver fet la comunicació amb `Send` i `Recv` no bloquejants, però hem decidit fer-ho així per assegurar-nos que no es continua executant el programa fins que s'envii tot i per tenir menys variables.

```
102  // Main loop: iterate until error ≤ tol a maximum of iter_max iterations
103  while ( error > tol && iter < iter_max ) {
104
105      if (rank>0){
106          //Li enviem la primera fila al proces anterior i rebem la ultima fila del proces anterior
107          MPI_Send(&A[m*first_row], m, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD);
108          MPI_Recv(&A[m*(first_row-1)], m, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD, &status);
109      }
110
111      if (rank<world_size-1){
112          //Li enviem la ultima fila al següent proces i rebem la primera fila del següent proces
113          MPI_Send(&A[m*final_row], m, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD);
114          MPI_Recv(&A[m*(final_row+1)], m, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD, &status);
115      }
116  }
```

Figura 4.

En quart lloc, hem modificat els paràmetres de la funció `laplace_step`. Ara, ha de començar el càlcul en la direcció de memòria de la matriu `A` més les files corresponents (depèn de cada procés). Aquest valor l'hem calculat prèviament i emmagatzemat en la variable `posicio_incial`, tal i com hem explicat en les inicialitzacions. D'igual forma, guardarem el

resultat en la matriu Anew, concretament en la fila inicial de cada procés. Per últim, indiquem que haurà de calcular les  $n_{\text{files}}$  que li toquin a cada procés amb  $m$  columnes. El resultat es pot observar en la Figura 5.

```
119 // Compute new values using main matrix and writing into auxiliary matrix
120 laplace_step(A+posicio_inicial, Anew+posicio_inicial, n_files, m);
```

Figura 5.

El següent canvi és igual al que acabem d'explicar però en la funció *laplace\_error*. Bàsicament hem de tornar a modificar els paràmetres per tal de calcular l'error correctament. Com l'error computat és un error parcial, hem fet un *MPI\_Allreduce*. Això significa que l'error obtingut és de cada procés, però hem de fer una comunicació per obtenir el màxim global que s'emmagatzemarà en la variable *error\_global*. Finalment, guardem el valor màxim en la variable *error* per no canviar les variables del bucle while, tal i com es veu en la Figura 6.

```
122 // Compute error = maximum of the square root of the absolute differences
123 error = laplace_error(A+posicio_inicial, Anew+posicio_inicial, n_files, m);
124 MPI_Allreduce(&error, &error_global, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
125 error = error_global;
```

Figura 6.

En sisè lloc, tornem a modificar les variables per copiar correctament la matriu Anew en A i poder fer les següents iteracions del bucle.

```
127 // Copy from auxiliary matrix to main matrix
128 laplace_copy(Anew+posicio_inicial, A+posicio_inicial, n_files, m);
```

Figura 7.

En acabar el bucle while, el procés 0 mesura el temps\_final amb la funció *MPI\_Wtime()*. El valor el guarda en la variable *t1*, prèviament inicialitzada. Per obtenir el temps total d'execució, fem una resta dels dos instants de temps. Finalment, alliberem memòria i tanquem la connexió MPI, com es veu en la Figura 8.

```
135 } // while
136
137 if (rank==0){
138     t1=MPI_Wtime();
139
140     printf("El temps d'execucio es: %f\n", t1-t0);
141 }
142
143 free(A);
144 free(Anew);
145
146 MPI_Finalize();
147 }
```

Figura 8.

## Resultat

Un cop teníem el codi acabat, hem decidit comprovar el rendiment d'aquest codi. S'ha fet mitjançant l'anàlisi de l'escalabilitat que té, és a dir, s'ha anat variant la quantitat de dades a processar o el nombre de recursos utilitzats. S'ha emprat dos models d'escalabilitat. En el primer model, strong scalability, el programa utilitza diferents mides de recursos, mantenint la mateixa mida de la matriu. La idea és que, si el sistema és perfectament escalable, l'eficiència haurà d'anar augmentant perquè conforme augmenten els processos, la càrrega de treball es divideix entre ells, reduint el temps d'execució. En el segon model, weak scalability, el programa va variant tant la mida d'entrada de la matriu com el nombre de recursos disponibles. La idea és que, si el sistema és perfectament escalable, l'eficiència hauria de romandre constant. Posteriorment es calcula el speedup comparant-ho amb el temps que triga el programa seqüencial.

### Strong Scaling

L'execució del programa s'ha dut a terme amb una mida del problema constant (4096x4096) i amb un màxim de 1000 iteracions. La Taula 1 mostra els temps d'execució obtinguts. Si els representem en una gràfica, obtenim la Figura 9.

Nº processos	Strong scalability	
	T(s) Wilma	Speedup Wilma
1	21.577454	1
2	12.453893	1.73
4	3.564716	6.05
8	2.650341	8.14

Taula 1.

Si analitzem els resultats de la Taula 1, observem una millora significativa en el temps d'execució a mesura que incrementem el nombre de processos. A més, l'augment en el speedup respecte el nombre de processos demostra una bona escalabilitat del codi amb el model strong scalability. Observem com els temps calculats són menors que els temps de referència donats en la pràctica. Això és pel fet que hem realitzat més paral·lisme, per tant, obtenim una solució més òptima, amb una major acceleració.

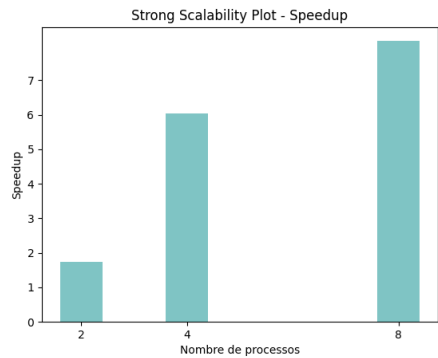


Figura 9.

Weak Scaling

L'execució del programa s'ha dut a terme amb una mida del problema variable i amb un màxim de 1000 iteracions. La Taula 2 mostra els temps d'execució obtinguts. Si els representem en una gràfica, obtenim la Figura 10.

Mida del problema	Nº processos	Weak scalability	
		T(s) Wilma	Efficiency Wilma
4096	1	21.577454	1
8192	2	12.701066	1.69
16384	4	14.169966	1.52
32768	8	21.743881	0.99

Taula 2.

Si analitzem els resultats de la Taula 1, observem que el codi obté una bona eficiència fins a 4 processos. Tot i això, aquesta disminueix quan aquest valor és 8. Això indica que el codi no és perfectament escalable amb el model weak scalability. Si observem la Figura 10, veiem com en realitat, el codi és eficient fins a 7 processos, ja que la línia blava és positiva en aquell moment. Concloïm, la millor eficiència s'obté amb dos processos.

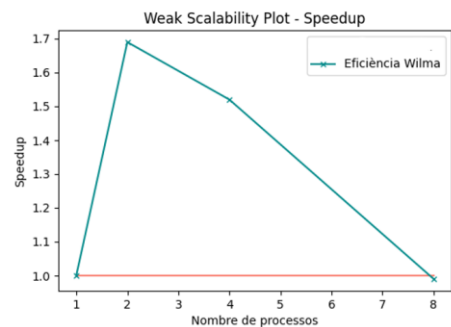


Figura 10.

En conclusió, mentre que l'escalabilitat forta és bona, l'escalabilitat dèbil no manté constant l'eficiència amb l'augment de recursos i mida del problema. La raó d'aquest empitjorament es deu a factors com la comunicació entre processos, que augmenta significativament respecte els recursos disponibles, donant lloc a un empitjorament en el rendiment.

## Principals problemes

Alguns dels problemes trobats a l'hora de fer la funció Laplace paral·lela s'expliquen a continuació.

Un problema que vam trobar va ser els errors de còpia que vam cometre en la comunicació entre els processos. Vam copiar i enganxar les files del Send i Recv i no vam actualitzar el valor d'una variable. Això va provocar que les dades no es transmetessin correctament entre els processos. Per solucionar aquest problema, ens va ser d'ajuda realitzar prints. D'aquesta forma visualitzàvem les posicions que estàvem calculant. A més, fer els dibuixos de la matriu i files respectives de cada procés van ser clau per entendre les iteracions de cada funció.

Un altre problema va ser a l'hora d'executar el codi. Al principi no vam limitar el càlcul a un sol procés. Per tant, els resultats no eren coherents. Tot i això, vam raonar el motiu perquè quan miràvem el rendiment el programa deixava de funcionar i vam corregir l'error.

Fialment, no sabíem com augmentar el número de processos. Ens sortia un error conforme el nombre d'aquests estava limitat a 2. Per solucionar-ho, vam examinar el fitxer *mpi.sub* i ens vam adonar que havíem de canviar el 2 la línia 4 per un 8, tal i com es veu en la Figura 11.

```
1 #!/bin/bash
2 #SBATCH --job-name=mpi_exec
3 #SBATCH -N 1 # number of nodes
4 #SBATCH -n 8
5 #SBATCH --distribution=cyclic
6 #SBATCH --partition=nodo.q
```

Figura 11.

Gràcies a aquestes millores, vam poder abordar amb èxit els problemes trobats en la implementació paral·lela de la funció Laplace.