Aránzazu Miguélez Montenegro 1587744
Emma Juanico Sala 1672960

INFORME PRÀCTICA MPI

Introducció i Plantejament del problema

En aquest informe s'explicarà les modificacions que s'han proposat per dur a terme la pràctica de paral·lelització amb l'entorn MPI, els resultats que s'han obtingut i els problemes que han anat sorgint.

La pràctica consisteix en un codi que resol l'equació 2D- Laplace amb el mètode iteratiu de Jacobi de manera seqüencial i el nostre objectiu és paral·lelitzar-lo amb la interfície de MPI per aconseguir un millor rendiment del programa i poder familiaritzar-nos amb les comandes per poder obtenir aquesta paral·lelització.

El codi seqüencial per poder resoldre l'equació consistia en una matriu 2D d'entrada, A, amb dimensions fixes n i m, i en el cas del nostre exemple, realitzarem un simple càlcul d'stencil on cada punt calcula el seu valor com la mitjana dels valors dels seus veïns. L'stencil està compost pel punt central i els quatre veïns directes. El càlcul s'itera fins que el canvi màxim de valor entre dues iteracions cau per sota d'algun nivell de tolerància o s'arriba a un nombre màxim d'iteracions.

Anàlisi del problema

El bucle més extern que controla el procés d'iteració s'anomena bucle de convergència. Aquest bucle itera fins que la resposta ha convergit, assolint una tolerància d'error màxim o un nombre d'iteracions.

El primer bucle niat dins del bucle de convergència hauria de calcular el nou valor per a cada element en funció dels valors actuals dels seus veïns. És necessari emmagatzemar aquest nou valor en una matriu diferent, o matriu auxiliar, que anomenem Anew. Si cada iteració emmagatzema el nou valor en si mateix, llavors existeix una dependència de dades entre els elements, ja que l'ordre en què cada element es calcula afecta la resposta final. Al emmagatzemar en una matriu temporal o auxiliar (Anew) ens assegurem que tots els valors es calculen utilitzant l'estat actual de A abans s'actualitzi. Com a resultat, cada iteració del bucle és completament independent una de l'altra. Aquestes iteracions de bucle es poden executar amb seguretat en qualsevol ordre i el resultat final serà el mateix.

Un segon bucle ha de calcular un valor d'error màxim entre els errors de cadascun dels punts de la matriu 2D. El valor d'error de cada punt de la matriu 2D es defineix com l'arrel quadrada de la diferència entre el nou valor (en Anew) i l'antic (en A). Si la quantitat màxima de canvi entre dues iteracions està dins

d'alguna tolerància, el problema es considera convergent i el bucle exterior acabarà.

El tercer bucle niat fa una actualització dels valors d'A amb els valors calculats a Anew. Si és l'última iteració del bucle de convergència, A tindrà els valors finals. Si el problema encara no ha convergit, llavors els valors d'A serviran com a dades d'entrada de la següent iteració.

Per últim, cada deu iteracions del bucle convergència s'ha d'imprimir l'error real a la pantalla per comprovar que l'execució avança correctament i l'error convergeix.

Disseny de la solució

Per poder paral·lelitzar el codi s'han hagut de fer una sèrie de modificacions que s'explicaran a continuació.

Les modificacions només s'han fet dins del main de la funció, és a dir, les funcions auxiliars (stencil, laplace_step, laplace_error, laplace_copy i laplace_init) no s'han canviat.

L'objectiu per poder paral·lelitzar és repartir la matriu entre els diferents processos disponibles i que cadascun faci els càlculs sobre una porció més petita que la matriu inicial. Per fer això, s'ha inicialitzat l'entorn MPI amb MPI_Init, MPI_Comm_Rank, MPI_Comm_Size i MPI_Status, que serveixen per inicialitzar, saber l'identificador del procés, el nombre de processos totals i l'estat d'una operació de comunicació.

Seguidament, s'ha calculat el nombre de files sobre els quals cada procés hauria de fer els càlculs i això és *num_files,* és a dir, el nombre total de files entre el nombre total de processos. També s'han determinat els índexs de les files inicials i finals per cada procés a *inicial* i *final* i per últim s'ha determinat l'índex inicial *init* per accedir a la matriu a memòria, ja que les dades es guarden en un array unidimensional.

Un cop incialitzades aquestes variables simplement reservem l'espai a memòria per a A i Anew i incialitzem la matriu A.

Ara, un cop dins del bucle, s'han de repartir a cada procés les seves files, però hem de tenir en compte que cada procés calcula *num_files* files i per fer els càlculs d'una certa posició es necessiten els 4 veïns directes, és a dir, la primera i última fila de cada procés no podran ser calculades si no tenim la fila anterior i posterior en cada cas. La primera fila de cada procés necessita la última fila del procés anterior i la última fila necessita la primera fila del següent procés. Excepcionalment, el procés 0 i l'últim són diferents, ja que al tenir la primera i

última fila de la matriu no poden tenir la fila prèvia i següent respectivament, així que s'haurà de tractar com a casos especials.

Així doncs, si no és el procés 0 s'envia la primera fila on es faria el càlcul al procés anterior i d'aquest mateix procés es rep la seva última fila, que seria l'anterior a la nostra primera. Amb això ja tenim solucionat el tenir la fila superior que es necessita per fer els càlcus, ara simplement s'ha de fer el mateix amb la fila inferior, és a dir, si no som l'últim procés, enviem la nostra darrera fila i rebem la primera fila del següent procés. Tot això s'emmagatzema a la matriu A.

Un cop tenim ja totes les files necessàries per fer els càlculs, els efectuem. Cada procés fa el *laplace_step* de la seva matriu A, és a dir, fa els càlcul corresponents utilitzant la matriu principal i els resultats els escriu a la matriu auxiliar. Seguidament, es calcula l'error d'aquests càlculs i un cop tenim tots els errors parcials de cada procés simplement s'ha de fer un reducció per buscar quin és el màxim d'aquest, indicant-li a la funció que volem trobar el màxim (MPI_MAX), i aquest *total* el guardem a la variable *error*.

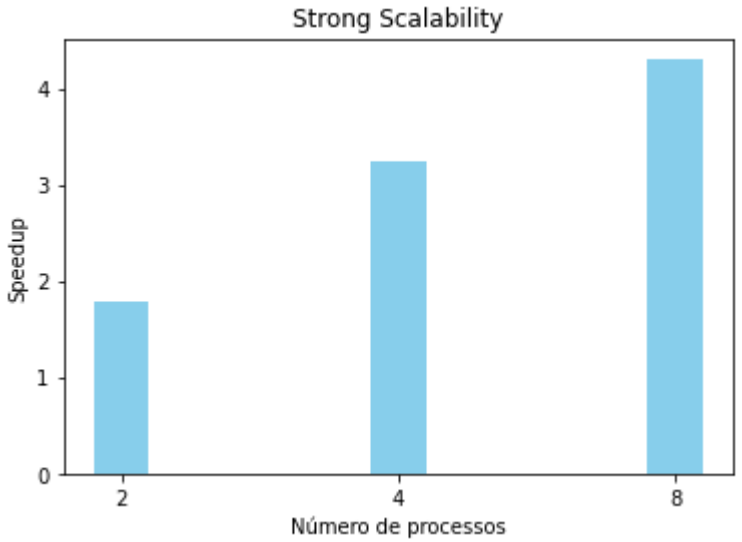
Per últim es copia els valors de la matriu auxiliar a la matriu principal, per poder tornar a fer el mateix procés.

Speedup

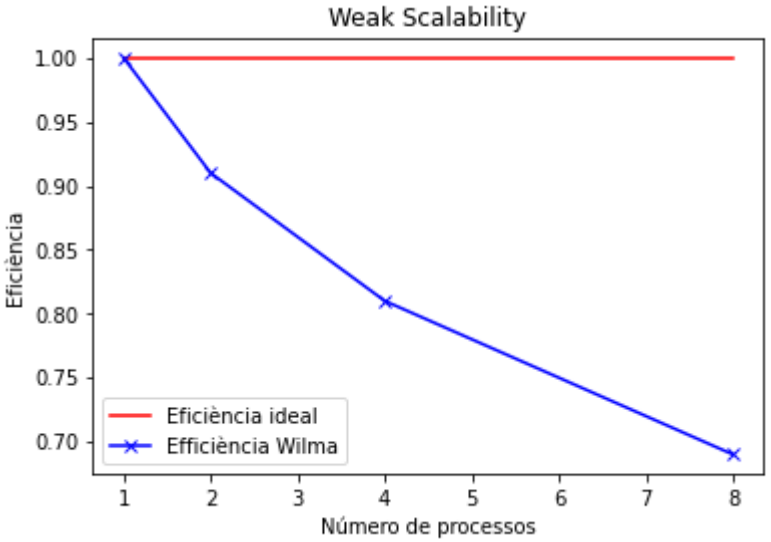
A continuació es mostren els resultats de les proves realitzades amb diferents mides de problema i número de processos al Wilma. Amb els resultats d'aquestes proves s'ha obtingut un cert speedup i eficiència que comentarem posteriorment.

	Strong scalability	
N° Processes (rows)	T(s) Wilma	Speedup Wilma
1	114,89	1,00
2	63,8316	1,8x
4	35,487	3,24x
8	26,688	4,3x

Aránzazu Miguélez Montenegro 1587744
Emma Juanico Sala 1672960



	Weak scalability		
Problem Size (rows)	N° Processes	Time(s)	Efficiency Wilma
4096	1	114,89	1
8192	2	125,818	0,91
16384	4	141,681	0,81
32768	8	165,793	0,69



Aránzazu Miguélez Montenegro 1587744
Emma Juanico Sala 1672960

Resultats

Després de paral·lelitzar el codi seqüencial del mètode de Jacobi per a l'equació de Laplace amb MPI, hem obtingut una sèrie de resultats que demostren l'eficàcia de la nostra aproximació. Com hem vist, es presenten els resultats de les proves realitzades tant per a escalabilitat forta com per a escalabilitat dèbil.

L'escalabilitat forta s'ha mesurat mantenint constant la mida del problema (4096x4096) i augmentant el nombre de processos.

Els resultats d'aquesta indiquen una millora significativa en el temps d'execució amb l'augment del nombre de processos. No obstant això, observem que el speedup no és lineal, cosa que es pot atribuir a la sobrecàrrega de comunicació entre processos.

Tot i això, el temps d'execució del codi seqüencial (núm. processos = 1) per a una mida de 4096x4096 i 1000 iteracions és de 114,89 segons. Comparat amb el codi paral·lelitzat, observem una reducció significativa en el temps d'execució, especialment amb un nombre més gran de processos.

L'escalabilitat dèbil l'hem mesurat augmentant la mida del problema de manera proporcional al nombre de processos. Hem vist que a mesura que augmentem la mida del problema i el nombre de processos, l'eficiència disminueix. Això pot ser degut a l'increment de la sobrecàrrega de comunicació i la latència associada a un major nombre de processos. Quan incrementem el nombre de processos, el *num_files* de cada procés disminueix. Això pot significar que cada procés té menys feina que fer entre comunicacions, cosa que augmenta la fracció de temps dedicat a la comunicació.

Problemes trobats

Tot i que podem observar una gran millora del nostre codi paral·lelitzat respecte el seqüencial, hi ha hagut certes dificultats per fer que aquest codi funcionés. El principal problema que ens hem trobat al resoldre aquesta pràctica és el repartiment de les files per cada procés. Des d'un bon principi vam entendre la necessitat que cada procés requeria les seves pròpies files i a més a més 2 files extres (excepte el primer i últim procés) per poder fer el càlculs, però a l'hora d'implementar aquest codi i els enviaments i rebudes vam tenir bastants problemes amb els índexs inicials i finals de cada procés i vam haver d'escriure-ho tot a un paper per entendre bé que havia de fer cada procés.

A més vam haver de trobar la manera de calcular els temps d'execució per tal de fer una anàlisi de rendiment del codi proposat. Això ens va suposar l'ús de la funció MPI_Wtime() la qual desconeixiem.

Aránzazu Miguélez Montenegro 1587744
Emma Juanico Sala 1672960

Conclusions

En conclusió, la paral·lelització del mètode de Jacobi amb MPI ha demostrat ser una eina poderosa per millorar el rendiment de càlculs intensius. Tot i els problemes trobats, com la gestió de la comunicació entre processos i la sobrecàrrega associada, els resultats indiquen que el codi paral·lelitzat és més ràpid que la seva versió seqüencial. Això ens permet executar problemes de major mida amb una eficiència raonable, fent ús òptim dels recursos de computació disponibles amb MPI.