Universitat Autònoma de Barcelona

Facultat de Ciències

Pràctica 3

Autors:

Gerard Lahuerta & Ona Sánchez

1601350 — 1601181

6 de Gener del 2023

1 Introducció

L'objectiu d'aquesta pràctica és optimitzar el temps d'execució del programa energystorm, programat al fitxer energy_storms.c.
Per tal d'optimitzar el funcionament del programa, es recurrirà a la paralel·lització de totes les instruccions repetitives, sempre i quan sigui possible, que generin un cost computacional elevat, al fitxer energy_storms_mpi.c.

2 Anàlisi del problema

A l'estudi realitzat anteriorment sobre els temps de diversos bucles del programa energy_storms.c[1], es va veure que la fase que més tarda en executar-se en la simulació és el bucle iniciat a la linia **183**, ja que el percentatge de temps que s'hi inverteix és casi del 100% del total del temps d'execució, pel que els esforços en optimitzar el codi haurien d'anar enfocats a aquest bucle.

Per altra banda, també es va concloure que els bucles iniciats a les línies **198** i **207** podrien ser paral·lelitzats, pel que s'ha d'estudiar quina és la millor manera d'optimitzar-los.

S'observen també bucles diversos que podrien unir-se en un de sol, com és el cas dels fors de les línies **175** i **176** (ja que usen el mateix rang de valors per l'índex k i no depenen entre ells).

Tot i així, la solució presentada tindrà un plantejament diferent a les presentades mitjançant *OpenMP* i *OpenACC*.

[1] L'estudi esmentat és el ja entregat anteriorment, si es vol consultar és: Estudi Pràctica 1 CAP

3 Disseny de la solució

Exposem i expliquem ara les seccions del codi modificades:

```
172 int rank, nprocs, size;
173 MPI_Status status;
174 MPI_Init( &argc, &argv );
175 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
176 MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
177
178 size = layer_size/nprocs;
179
180 float *layer = (float *)malloc( sizeof(float) * layer_size );
181 float *layer_copy = (float *)malloc( sizeof(float) * layer_size );
182 float *layer_priv = (float *)malloc( sizeof(float) * size );
183 float *layer_copy_priv = (float *)malloc( sizeof(float) * size );
184
185 if ( layer == NULL || layer_copy == NULL ) {
186     fprintf(stderr,"Error: Allocating the layer memory\n");
187     exit( EXIT_FAILURE );
188 }
189
190 if(rank == 0){
191      #pragma omp parallel for
192      for( k=0; k<layer_size; k++ ){ layer[k] = 0.0f; layer_copy[k] = 0.0f; }
193 }
194
195 MPI_Scatter(layer, size, MPI_FLOAT, layer_priv, size, MPI_FLOAT, 0, MPI_COMM_WORLD);
196 MPI_Scatter(layer_copy, size, MPI_FLOAT, layer_copy_priv, size, MPI_FLOAT, 0,
197             MPI_COMM_WORLD);
```

S'inicien els processos paral·lels amb MPI i s'utilitza el mètode *Scatter* per a dividir l'inicialització dels vecctor *layer* i *layer_copy* duts a terme pel procès amb rank 0 (el master).

A més, s'inicialitcen els vector auxiliars de cada procès amb el tamany que hi necesiten per a poder treballar ($size = \frac{layer\_size}{nprocs}$).

Els vectors que utilitzarà cada procès per a fer els càlculs són: *layer_priv* i *layer_copy_priv*.

```
209 for( j=0; j<storms[i].size; j++ ) {
210     float energy = (float)storms[i].posval[j*2+1] * 1000;
211     int position = storms[i].posval[j*2];
212 
213     #pragma omp parallel for
214     for( k=0; k<size; k++ ) {
215 
216         int distance = abs(position - (rank*size + k)) + 1;
217         float atenuacion = sqrtf( (float)distance );
218 
219         float energy_k = energy / (layer_size * atenuacion);
220         float Tls = THRESHOLD / layer_size;
221 
222         if ( energy_k >= Tls || energy_k <= -Tls )
223                 layer_priv[k] = layer_priv[k] + energy_k;
224 
225     }
226 }
```

El següent fragment de codi modificat és el càlcul de l'energia de les cel·les.

S'ha eliminat la funció *update* i s'ha inserit directament el codi en el bucle for.

A més, els càlculs ara es fan sobre el vector propi de cada procès.

```
235 #pragma omp parallel for
236 for( k=0; k<size; k++ ) layer_copy_priv[k] = layer_priv[k];
237 float inferior, superior;
238
239 if(rank != nprocs-1){
240     MPI_Send(&layer_priv[size-1], 1, MPI_FLOAT, rank + 1, rank, MPI_COMM_WORLD);
241     MPI_Recv(&superior,1,MPI_FLOAT, rank+1, rank+1+nprocs, MPI_COMM_WORLD, &status);
242 }
243 if(rank != 0){
244     MPI_Send(&layer_priv[0], 1, MPI_FLOAT, rank - 1, rank + nprocs, MPI_COMM_WORLD);
245     MPI_Recv(&inferior, 1, MPI_FLOAT, rank - 1, rank - 1, MPI_COMM_WORLD, &status);
246 }
```

Expliquem ara l'enviament d'informació entre processos:

- Primer if: $rank! = nprocs - 1$

  Tots els processos que no siguin l'últim envien l'últim element del seu propi vector *layer_priv* al següent procès; és a dir, el procès de rank 1 envia l'últim element al procès de rank 2 i així successivament.

- Segon if: $rank! = 0$

  Tots els processos que no siguin el primer envien el primer element del seu propi vector *layer_priv* al procès anterior; és a dir, el procès de rank 1 envia el primer element al procès de rank 0 i així successivament.

Aquesta informació enviada es guarda a les variables *superior* i *inferior* (si és l'últim element que s'envia o el primer que s'envia respectivament) i s'utilitzarà per a calcular l'energia de les celdes.

```
252 if(rank == 0){
253     #pragma omp parallel for
254     for ( k=1; k<size; k++ ) {
255         if (k != size-1) layer_priv[k] =
256             (layer_copy_priv[k-1]+layer_copy_priv[k]+layer_copy_priv[k+1])/3;
257         else layer_priv[k] = (layer_copy_priv[k-1]+layer_copy_priv[k]+superior)/3;
258     }
259 }
260 else if(rank == nprocs -1){
261     #pragma omp parallel for
262     for( k=0; k<size-1; k++ ) {
263         if(k != 0) layer_priv[k] =
264             (layer_copy_priv[k-1]+layer_copy_priv[k]+layer_copy_priv[k+1])/3;
265         else layer_priv[k] = (inferior+layer_copy_priv[k]+layer_copy_priv[k+1])/3;
266     }
267 }
268 else {
269     #pragma omp parallel for
270     for( k=0; k<size; k++ ){
271         if(k == 0) layer_priv[k] =
272             (inferior+layer_copy_priv[k]+layer_copy_priv[k+1])/3;
273         else if(k == size - 1) layer_priv[k] =
274             (layer_copy_priv[k-1]+layer_copy_priv[k]+superior)/3;
275         else layer_priv[k] =
276             (layer_copy_priv[k-1]+layer_copy_priv[k]+layer_copy_priv[k+1])/3;
277     }
278 }
```

Es calcula l'energia de cada celda mitjançant la informació enviada entre els processos, distingint els casos de l'últim i el primer procès, ja que el procès de càlcul és diferent als que es troben entremig.

```
275 struct tupla {
276     float maxi_layer;
277     int posicio_layer;
278 };
279
280 struct tupla *aux = malloc(size * sizeof *aux);
281 struct tupla *maxi = malloc(size * sizeof *maxi);
282
283 #pragma omp parallel for
284 for(k = 0; k < size; k++){ aux[k].maxi_layer = layer_priv[k];
285                            aux[k].posicio_layer = rank*(size) + k;}
286
287 MPI_Reduce(aux, maxi, size, MPI_FLOAT_INT, MPI_MAXLOC, 0, MPI_COMM_WORLD);
288 maximum[i] = maxi[0].maxi_layer;
289
290 if(rank == 0){
291     for(j=1; j<size; j++){
292         if(maxi[j].maxi_layer > maximum[i]){
293                 maximum[i] = maxi[j].maxi_layer;
294                 positions[i] = maxi[j].posicio_layer;
295         }
296     }
297 }
```

Es crea una estructura que hi guardi la prosicció (real) de la celda i el valor de l'energia de cada vector dels diversos processos.
Mitjançant el mètode *Reduce* agrupem el valor màxim i la seva posició de cada procès en el procès amb rank 0 que s'encarregarà de trobar el màxim etre ells i guardar el seu valor i la seva posició a les variables *maximum* i *position*.

S'ha simplificat la paral·lelització de la millor manera possible per tal d'optimitzar al màxim el codi. Informar que les comandes d'OpenMP que es veuen en el codi son degudes a un estudi posterior on s'ha intentat millorar l'optimització implementant a la vegada OpenMP.

L'anàlisis d'aquest estudi és explicats posteriorment.

En cas de voler consultar les zones modificades, en les seccions de codi hi ha indicades les files on es troben les comandes al fitxer energy_storm_mpi.c.

4 Resultat

	TEMPS			
	SEQÜENCIAL	PARAL·LELITZAT		
		2 processadors	4 processadors	6 processadors
Test 2	43.369	46.931	24.323	16.245
Test 7	241.569	262.889	132.782	89.083
Test 8	5.260	12.337	8.516	7.246
	Acceleració			
Test 2	-	0.924	1.783	2.669
Test 7	-	0.918	1.819	2.711
Test 8	-	0.426	0.617	0.725

S'observa com s'ha optimitzat el programa obtenint una millora de fins a 2.711 vegades més ràpid amb 6 processos en el test 7.

Concluïm que hem assolit els objectius millorant les zones que ja s'havien estudiat anteriorment, en els casos dels tests 2 i 7, mentre que el test 8 no s'ha aconseguit reduir el temps de còmput.

Destacar que en el cas d'usar únicament 2 processadors per distribuïr la feina no s'aconsegueix accelerar l'execució degut a que l'alt cost de comunicació entre processos no es veu compensat per la distribució del treball, cosa que si que es veu compensada en el cas d'usar més processadors.

5 OpenMP

S'ha estudiat també la millora del temps de còmput en el cas d'afegir comandes d'OpenMP a certs bucles del programa energy_storms_mpi.c, optimitzat anteriorment usant MPI. Els resultats es mostren a continuació:

	TEMPS			
	SEQÜENCIAL	PARAL·LELITZAT		
		2 processadors	4 processadors	6 processadors
Test 2	43.369	45.694	23.553	15.379
Test 7	241.569	267.853	131.307	87.903
Test 8	5.260	12.313	8.733	7.269
	Acceleració			
Test 2	-	0.949	1.841	2.820
Test 7	-	0.901	1.839	2.748
Test 8	-	0.427	0.602	0.723

S'observa com s'ha optimitzat el programa obtenint una millora de fins a 2.820 vegades més ràpid amb 6 processos en el test 2.

Es dedueix també que l'ús d'OpenMP disminueix el temps d'execució del programa en la gran majoria dels casos, tot i que la millora resulta ser poc significativa, sent la major millora de 0.866 segons.

Concluïm, d'igual manera que anteriorment a 4, que hem assolit els objectius millorant les zones que ja s'havien estudiat anteriorment, en els casos dels tests 2 i 7, mentre que el test 8 no s'ha aconseguit reduir el temps de còmput, i que els casos on no s'ha aconseguit una millora dels temps és degut a l'alt cost de la comunicació entre processos.

6 Comparativa de mètodes

Es mostren a continuació els millors temps i acceleracions aconseguides per cada test, amb el seu mètode corresponent. Si es desitja examinar les dades sobre els mètodes OpenMP i OpenACC, es poden consultar als estudis entregats anteriorment a: Pràctica 1 CAP i Pràctica 2 CAP.

	Temps	**Millor temps**	**Acceleració**	**Mètode**
	SEQÜENCIAL	PARAL·LELITZAT		
Test 2	43.369	3.865	11.220	OpenACC
Test 7	241.569	20.755	11.639	OpenMP (12 fluxos)
Test 8	5.260	1.275	4.125	OpenMP (12 fluxos)

Es dedueix de la taula anterior que el mètode que aconsegueix una millor acceleració és la nostra implementació mitjançant l'OpenMP que, com més fluxos utilitzi, més permet reduir el temps de còmput, aconseguint una acceleració més alta, de manera que els millors temps trobats han sigut usant el màxim nombre de fluxos estudiats, 12.

S'observa que l'únic test pel que la millor opció no és usar OpenMP és el test 2, que aconseguia una acceleració màxima de 10.340, davant 11.220 assolit amb la implementació proposada mitjançant OpenACC.

Es mostra a continuació els càlculs d'eficiència de cada mètode:

[redacted]	EFICIENCIA							
	OpenMP				OpenACC	MPI		
	2 fluxos	4 fluxos	6 fluxos	12 fluxos		2 proc.	4 proc.	6 proc.
Test 2	0.981	0.970	1.231	0.861	0.087	0.474	0.460	0.470
Test 7	0.986	0.983	1.300	0.969	0.087	0.450	0.459	0.458
Test 8	0.842	0.645	0.624	0.343	0.001	0.213	0.150	0.120

S'observa com la implementació més eficient és la que utilitza *OpenMP*, tot i no ser la que millor acceleració obté en tots els tests. Per aquest motiu, es conclou i es recomana utilitzar la implementació que utilitza *OpenMP*.

Per altra banda, si es vol obtenir una bona acceleració ignorant l'eficiència del programa, es recomana utilitzar l'implementació amb *OpenACC* ja que obté resultats similars o millors que la optimització que implementa *OpenMP*.

Per altra banda, si l'estudi conté grans quantitats de dades; tot i no haver obtingut molts bons resultats amb els tests, es recomana utilitzar l'optimització amb *MPI* ja que el cost de repartiment i enviament de dades pot ser rentable i, amb la modificació que inclou a més *OpenMP* pot obtenir molts bons resultats. Tot i així aquesta conclusió és purament intuitiva ja que no s'ha testejat, s'extreu dels resultats obtinguts en les taules.

7 Principals problemes

Enumerem ara els principals problemes que han aparegut en el nostre procès d'optimitzacdió del codi:

1. Optimització del bucle de la linia 252:

   Vam tenir bastants problemes al plantejar la paral·lelització dels càlculs de cada procès ja que no trobavem la forma correcta de distribuïr les dades; tot i així, una vegada provades varies idees (utilitzar *Scatter*, enviar més dades, etc), es va decidir simplificar el mètode enviant només els elements que es necesiten per a calcular l'energia de les celdes que no es trobaben als altres processos.

2. Optimització del bucle de la linia 275:

   El problema principal era com enviar les dades necessaries, primerament es va plantejar de fer la cerca de forma manual però el procès era molt ineficient; pel que cercant en els apunts, a internet i comentant estrategies/ideees dels companys es va probar a fer una estructura i utilitzar el mètode *Reduce* per a enviar la informació al node principal, on s'acabaria de fer la cerca.