Lab 1 - OpenMP Report

Nom: Gorka, Arturo i Ferran

Grup: G14

Assignatura: Introducció a la Programació Paral·lela i Distribuïda.

Data: 20/04/2025

Objectiu del laboratori

Aquest laboratori té com a objectiu practicar amb **OpenMP**, una API per a la programació paral·lela sobre sistemes de memòria compartida. A través de tres exercicis, es pretén entendre com millorar el rendiment dels programes seqüencials mitjançant tècniques de paral·lelització: multithreading, sincronització, i control de tasques.

En aquesta pràctica hem completat:

- **1. Factorització de Cholesky** (versió seqüencial i paral·lela)
- **2. Càlcul d'histograma** (4 mètodes de paral·lelització)
- **3. Càlcul d'Argmax** (4 estratègies seqüencials i paral·leles)

Exercici 1: Factorització de Cholesky

Apartat 1: Expose your parallelization strategy to divide the work in the Cholesky algorithm and in the matrix multiplication. Justify the selection of the scheduler and chunk size and compare different schedulers with different chunk sizes and show the results.

Descripció general

A partir d'una matriu simètrica definida positiva **A**, es calcula la seva factorització de Cholesky, és a dir: A = L*U = L*Lt on L és triangular inferior i U = L^T. Un cop feta la factorització, es comprova que LU = A amb un marge d'error inferior al 0.001%.

Paral·lelització i estrategies

1. **Inicialització de matrius:** paral·lelitzada completament perquè cada cel·la és independent.

```
#pragma omp parallel for private(j)
for(i=0; i < n; i++) {
   for(j=0; j < n; j++) {
      L[i][j] = 0.0;
      U[i][j] = 0.0;
```

```
    }
}
```

2. **Càlcul de valors diagonals de U:** schedule(static) amb reducció sobre tmp, ja que totes les iteracions tenen temps de càlcul similar.

```
// Diagonals
#pragma omp parallel for reduction(+:tmp) schedule(static)
for(k=0;k<i;k++) {
    tmp += U[k][i]*U[k][i];
}
```

3. **Valors no diagonals:** schedule(dynamic, 16) per compensar l'increment de càlcul en iteracions amb valors alts de i.

```
// NO diagonals
#pragma omp parallel for private(j,k,tmp) schedule(dynamic, 16)
for(j=i+1;j<n;j++) {
    tmp = 0.0;
    for(k=0;k<i;k++) {
        tmp += U[k][i]*U[k][j];
    }
    U[i][j] = (A[i][j]-tmp)/U[i][i];
}
```

3. **Transposició:** schedule(static) o dynamic(8) segons mida. Iteracions regulars.

```
#pragma omp parallel for private(i, j) schedule(static)
for(i=0;i<n;i++) {
    for(j=0;j<=i;j++) {
        L[i][j] = U[j][i];
    }
}
```

4. **Multiplicació B = LU:** schedule(dynamic, 8) degut a la variabilitat d'iteracions útils (condició if).

```
#pragma omp parallel for private(i, j, k) schedule(dynamic, 8)
for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
        B[i][j] = 0.0;
        for(k=0;k<n;k++) {
            if (i >= k && k <= j) { // Only use non-zero elements of L and U
                B[i][j] += L[i][k] * U[k][j];
            }
```

```
      }
    }
  }
```

Justificació de les estratègies.

Programació Estàtica:

- S'utilitza per a tasques amb una distribució uniforme de la càrrega de treball (inicialització, transposició).
- Funciona bé per a bucles on cada iteració requereix aproximadament el mateix temps.
- Poc overhead, ja que la divisió de la feina es determina abans de l'execució.
- Òptim per al càlcul dels elements diagonals i l'operació de transposició.

Programació Dinàmica:

- S'utilitza per a tasques amb càrrega de treball irregular (elements no diagonals de Cholesky, multiplicació de matrius).
- Millor on les iteracions poden tenir temps d'execució variables.
- Major overhead degut a la distribució de la feina en temps d'execució.
- Essencial per a elements no diagonals i multiplicació de matrius on la càrrega de treball varia.

Selecció de la Mida de Chunk

Mida de Chunk 16 per als Càlculs No Diagonals:

- Les iteracions del bucle tenen una longitud variable (depèn del valor de *i*).
- Mida de chunk mitjana que equilibra l'overhead de la programació amb la distribució de la càrrega.
- A mesura que *i* augmenta, cada iteració esdevé més intensiva en càlculs.

Mida de Chunk 8 per a la Multiplicació de Matrius:

- Mida de chunk més petita per un millor equilibratge de càrrega a un nivell més detallat.
- La càrrega de treball de la multiplicació de matrius és més irregular a causa de la comprovació condicional.
- Chunks més petits asseguren una millor distribució de la càrrega computacional més pesada.

Comparativa de rendiment

Per tal d'afirmar aixo hem executat el programa amb diferentes configuracions de tamany de chunk:

Planificador	Chunk	Tiempo Cholesky (s)	Tiempo B=LU (s)
static	1	3.77	1.64
static	4	**0.70**	**1.49**
static	8	1.13	**1.49**
static	16	2.86	1.78
static	32	**0.71**	1.74
dynamic	16	2.08	1.80

Conclusions:

- Cholesky: `static` amb chunk 4 o 32 da el millor rendimient.
- Multiplicació: `static` con chunk 4 o 8 son óptims.
- El `schedule(dynamic)` te més overhead, pero pot ser útil si hiha variabilidad forta, pero aquí el `static` ben ajustat rindeix més.

Apartat 2: ***Make two plots: one for the speedup of the Cholesky factorization and another for the matrix multiplication for n " 3000. Use 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results.***

Escalabilitat i Speedup

Speedup (string scaling) de les dues fases principals amb n = 3000:

Cholesky:

Fils	Temps Secuencial (s)	Temps Paral·lel (s)	Speedup	Eficiència (%)
1	9.27	9.00	1.03	103.0
2	8.92	4.38	2.04	102.0
4	8.89	2.43	3.66	91.5
8	8.83	1.30	6.79	84.9
16	8.85	2.08	4.25	26.6

B = LU:

Fils	Temps Secuencial (s)	Temps Paral·lel (s)	Speedup	Eficiència (%)
1	18.27	18.17	1.01	101.0
2	18.62	9.27	2.01	100.5
4	18.65	5.85	3.19	79.8
8	18.47	3.07	6.02	75.3
16	18.41	1.80	10.23	64.0

Anàlisis:

Cholesky:

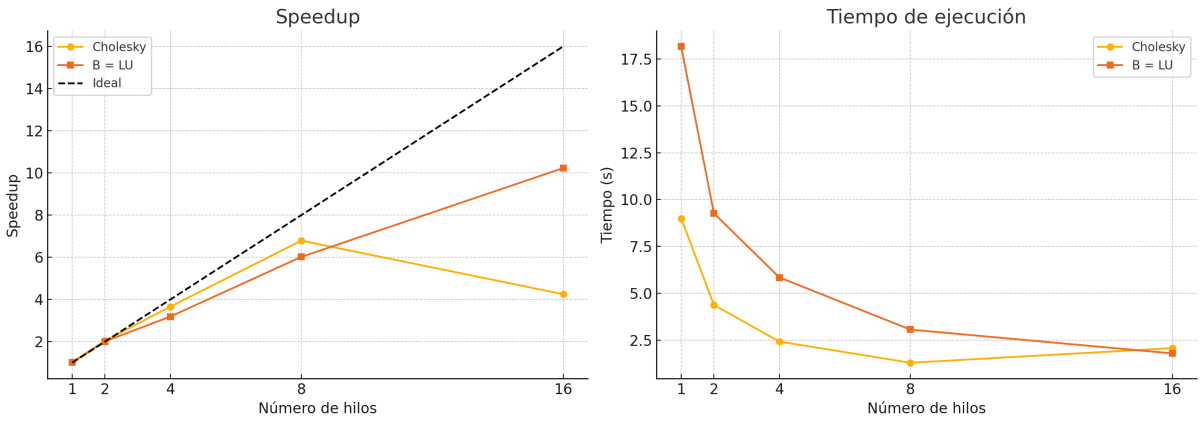
- Escala bé fins a 8 fils.
- El rendiment cau a partir de 16 fils, probablement per contenció de memòria i sincronització en càlculs diagonals.
- L'eficiència es desploma fins al 26.6% amb 16 fils, indicant un límit en el paral·lelisme útil.

Multiplicació B = LU:

- Mostra millor escalabilitat i eficiència (fins al 64% amb 16 fils).
- Això és degut a la seva menor dependència de dades i millor paral·lelisme natural.

Recomanacions finals

- Per a **Cholesky**, utilitzar `schedule(static, 4)` i un màxim de **8 fils**.
- Per a **B = LU**, es poden utilitzar **fins a 16 fils** amb `schedule(dynamic, 8)` per una bona distribució.
- Es pot explorar la tècnica de **blocking** per millorar l'ús de la memòria cau i reduir la contenció.
- Comparar sempre amb el **speedup ideal** (línia recta) per avaluar com d'allunyats estem del paral·lelisme perfecte.



Exercici 2: Histograma

Descripció general:

Tenim un vector de N = 1.000.000 valors entre 0 i 49. Cal construir el seu histograma de freqüências. Hem implementat i comparatistas 4 estratègies:

Apartat 1: Explain how have you solved each of the parallelizations.

Implementacions:

- **Paral·lelització amb critical:**

Hem utilitzat #pragma omp critical per assegurar que només un fil actualitzi l'histograma alhora. Això garanteix la correcció, però limita el rendiment en paral·lel, ja que la secció crítica es converteix en un bottleneck, ja que només un fil hi pot accedir alhora.

- **Paral·lització amb atomic:**

Hem fet servir *#pragma omp atomic* per actualitzar cada casella de l'histograma. Això és més eficient que critical perquè introdueix menys sobrecàrrega, encara que segueix serialitzant l'accés a les mateixes posicions de memòria. Funciona bé quan les col·lisions no són freqüents.

- **Paral·lelització amb locks:**

Hem fet servir un array d'omp_lock_t, un per a cada casella de l'histograma. Els fils només bloquegen la casella que necessiten actualitzar, de manera que la contenció es redueix significativament en comparació amb una secció crítica global. Això proporciona millor paral·lelisme, encara que comporta la sobrecàrrega de gestionar els bloquejos.

- **Paral·lelització amb reduction:**

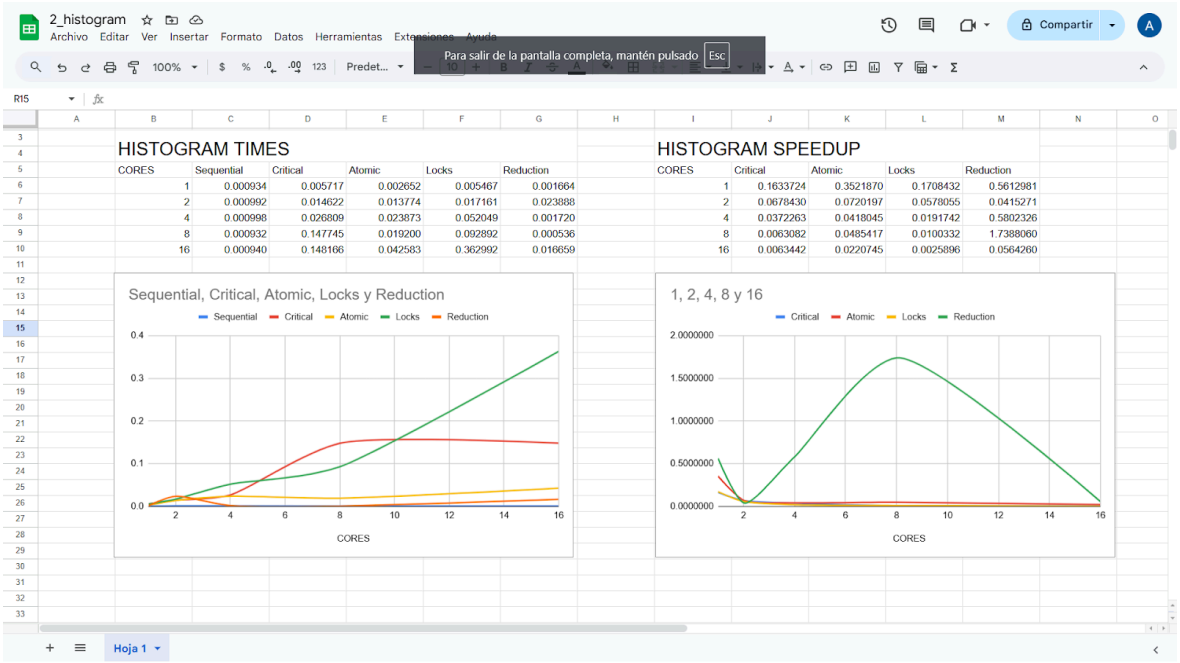
Hem implementat una reducció manual: cada fil construeix el seu propi histograma privat i, en acabar, els resultats parcials es combinen. Això evita sincronització durant el bucle principal, oferint el millor rendiment, especialment quan hi ha molts fils i col·lisions.

Apartat 2: Explain the time differences between different parallel methods if there are any.

Mètode	Descripció	Avantatges	Desavantatges
Critical	Bloqueja una secció crítica completa per a accés exclusiu.	Fàcil d´implementar, garanteix seguretat de dades.	Baix rendiment amb molts fils.
Atomic	Assegura operacions atòmiques sobre una variable compartida.	Menys overhead que Critical, millor per a operacions simples.	Hi continua havent col·lisions si molts fils accedeixen al mateix valor.
Locks	Usa un lock per protegir accessos concurrents.	Permet un control per cada element, més flexible.	Molt overhead per lock/unlock, escalabilitat limitada.
Reduction	Cada fil treballa a una còpia local i es combina al final.	Minimitza sincronització, excel·lent escalabilitat.	Ús extra de memòria per a les còpies locals.

Apartat 3: Make a speedup plot for the different parallelization methods for 1, 2, 4, 8, and 16 cores. Discuss the results.

El gràfic de l'esquerra mostra el temps que ha trigat, i el de l'esquerra el speedup obtingut en paral·lelitzar el càlcul de l'histograma usant Critical, Atomic, Locks i Reduction, comparats sobre 1, 2, 4, 8 i 16 cors.



Observacions:

- **Reduction** és la tècnica que clarament aconsegueix el millor rendiment, especialment a 8 cors, reduction minimitza la sobrecàrrega de sincronització en acumular resultats localment i combinar només al final.
- **Critical, Atomic y Locks** mostren un speedup baix i fins i tot degraden a mesura que s'hi afegeixen més cors. Això és degut al coll d'ampolla per sincronització: tots els fils competeixen constantment per accés a la mateixa variable compartida.
- **Locks** és el que pitjor escala, mostrant fins i tot temps majors que la versió seqüencial quan s'usen molts fils, a causa de la sobrecàrrega
- **Atomic** és millor que **Critical** y **Locks**, però tot i així el seu speedup és limitat perquè cada actualització continua sent una operació bloquejant

El gràfic demostre que:

- Mètodes amb alta sincronització (**Critical**, **Locks**) penalitzen molt quan l'accés concurrent a dades és intensiu.
- Mètodes amb acumulació local com a **Reduction** permeten paral·lelisme efectiu, en reduir la necessitat de sincronització.
- No sempre més cores vol dir més velocitat.

Exercici 3: Argmax

Aquest exercici té com a objectiu calcular el màxim d'un vector de float i la seva posició (índex) utilitzant diferents estratègies de programació seqüencial i paral·lela amb **OpenMP**.

Apartat 1: Explain the different implementations of the argmax function (sequential and recursive),/ and how you parallelized each of them.

Funcions implementades

1. **Versió seqüencial (argmax_seq)**

- Recorrer el vector una sola vegada utilitzant un for
- En cada iteració, compara el valor actual amb el màxim i actualiza m i idx_m si troba un major.
- No hi ha paral·lelització.

2. **Versió paral·lela amb for (argmax_par)**

- Dividiu el treball entre fils usant #pragma omp for.
- Cada fil troba el màxim local i el seu índex.
- Després, en una secció crítica (#pragma omp critical), cada fil intenta actualitzar el màxim global si el seu és més gran.
- Limitació: la secció crítica pot generar colls d'ampolla si hi ha molts fils.

3. **Versió recursiva seqüencial (argmax_recursive)**

- Divideix el vector en meitats fins que la mida sigui menor o igual a K.
- En aquest punt base, s'anomena argmax_seq.
- Després, combina els resultats de totes dues meitats comparant els màxims.
- No hi ha paral·lelització, però permet comparar estructura recursiva vs iterativa.

4. Versió recursiva amb tasques (argmax_recursive_tasks)

Igual a la versió recursiva, però cada crida a una meitat es llança com una tasca:

- Se sincronitza amb #pragma omp taskwait.
- En fer servir #pragma omp single en main, es llança la primera tasca i es creen totes les altres des d'allà.
- És la versió amb més paral·lelisme potencial.

Apartat 2: Run the code with 2, 4 and 8 threads for a vector of size N " 4096 ˚ 4096 and plot the strong speedup for both parallel implementations. For the recursive and tasks implementations plot the a strong speedup curve for the following values of K, K " 16, 512, 2048, 4096 and K " 8192. Include all curves in the same plot for better comparison. Comment on the obtained results, and provide an explanation for the behaviors of the different parallel implementations. Hint: depending on the node load, the results might

vary. Launch the job several times and keep the cases with the largest speedups. If well implemented, the tasks version should be at least two times faster using K " 8192 and 8 threads.

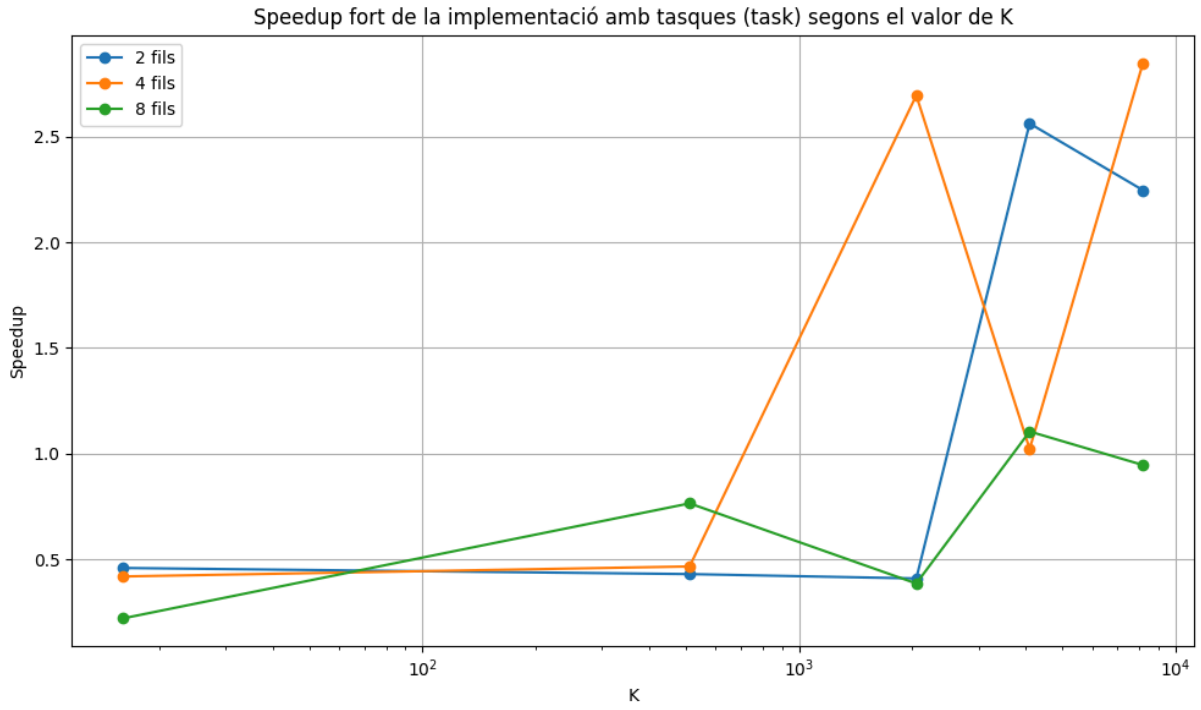
Dades utilitzades

Per a aquest exercici s'han fet proves amb mides de vector N=4096×4096N = 4096 \times 4096N=4096×4096 i diferents valors de KKK: 16, 512, 2048, 4096, 8192.

S'han comparat dues implementacions paral·leles: `for` i `task`, utilitzant 2, 4 i 8 fils.

Gràfica: Speedup fort de la implementació amb tasques segons KKK

Inclourem totes les corbes de `task_speedup` per comparar els efectes de variar el valor de KKK.



Comentari dels resultats

- Per a KKK petit (16, 512): el speedup és baix, sovint inferior a 0.5. Això indica que el cost de crear tasques i la falta de paral·lelisme suficient penalitzen el rendiment.
- Per a K=2048K = 2048K=2048: comença a veure's un augment notable del speedup, especialment amb 4 fils (fins a 2.69).
- Amb K=4096K = 4096K=4096: el comportament és irregular. Amb 2 fils s'obté un gran speedup (>2), però amb 4 i 8 baixa o es manté estable.
- Amb K=8192K = 8192K=8192: la millor configuració és amb 4 fils (speedup 2.84), però sorprenentment amb 8 fils el speedup baixa (0.94), probablement per sobrecàrrega i poca eficiència de tasques petites.

> Conclusió: El millor rendiment amb tasques s'obté amb valors alts de KKK, però també depèn molt del nombre de fils. Amb 4 fils s'obté la millor escalabilitat general. Amb 8 fils, el sistema pot patir més per la sobrecàrrega.

Apartat 3: What is the arithmetic intensity of the argmax algorithm? Which resource (memory bandwidth or peak computing capacity) do you expect to be the bottleneck for throughput?

Intensitat aritmètica

L'algorisme `argmax` realitza una única operació (comparació) per a cada accés a un element del vector.

- Operacions: 1 comparació per element
- Accessos a memòria: 1 lectura per element

Intensitat aritmètica = Operacions flotants / Accesos a memòria

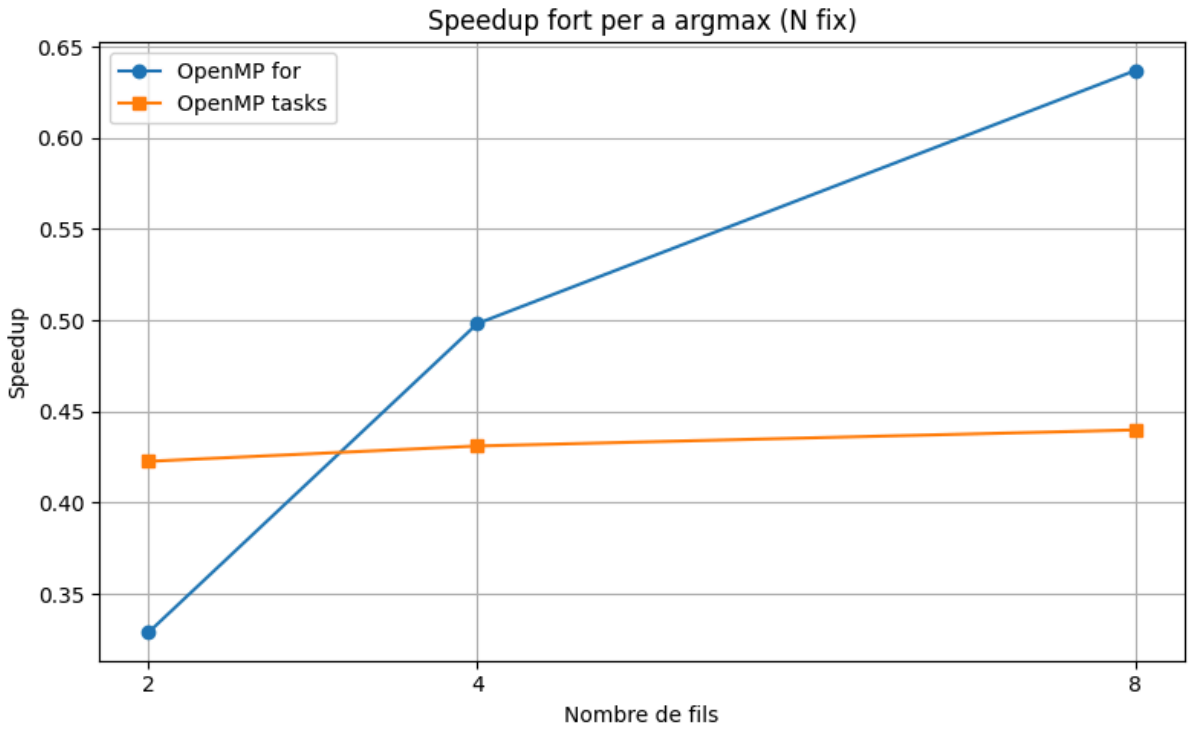
→ **IA = 1 / 1 = 1 FLOP/accés**

Anàlisi del bottleneck

Amb una intensitat aritmètica tan baixa (1), l'algorisme és **clarament limitat per l'amplada de banda de memòria**, no per la capacitat de càlcul del processador. Això significa que:

- El rendiment no millorarà gaire augmentant els fils si l'accés a memòria no es pot paral·lelitzar eficientment.
- Les optimitzacions haurien de centrar-se en **millorar la localitat de memòria** o reduir els accessos redundants.

Gràfica: Speedup per a `for` i `task` amb N fix



Comentari dels resultats

- Tant `for` com `task` mostren **speedups molt modestos**.
- La versió amb `task` no millora amb més fils, i fins i tot empitjora lleugerament.
- La versió amb `for` escala una mica millor, però mai arriba a 1.

> Conclusió: Aquest algorisme no escala bé amb paral·lelisme per la seva baixa intensitat aritmètica. El bottleneck és la memòria, no la CPU. Cal optimitzar l'accés a dades més que el càlcul.

- Dividiu el vector en meitats fins que la mida sigui menor o igual a K.
- En aquest punt base, s'anomena argmax_seq.
- Després, combina els resultats de totes dues meitats comparant els màxims.
- No hi ha paral·lelització, però permet comparar estructura recursiva vs iterativa.