REPORT LAB 1

1. Cholesky

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.

PART1: El nostre programa realitza una descomposició de Cholesky d'una matriu definida simètrica i positiva. Hi ha dues versions de l'algoritme: la seqüencial i la paral·lelitzada amb OpenMP.

Per l'algoritme paral·lelitzat, primer hem inicialitzat les matrius (A, U, L, B). A està inicialitzada amb valors entre -1 i 1 i és convertida en una matriu positiva, mentre que L i U estan inicialitzades amb valor 0.

```
/**
 * 1. Matrix initialization for A, L, U and B
 */
start = omp_get_wtime();
A = (double **)malloc(n * sizeof(double *));
L = (double **)malloc(n * sizeof(double *));
U = (double **)malloc(n * sizeof(double *));
B = (double **)malloc(n * sizeof(double *));

for(i=0; i<n; i++) {
    A[i] = (double *)malloc(n * sizeof(double));
    L[i] = (double *)malloc(n * sizeof(double));
    U[i] = (double *)malloc(n * sizeof(double));
    B[i] = (double *)malloc(n * sizeof(double));
}

srand(time(NULL));
// Generate random values for the matrix
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        A[i][j] = ((double) rand() / RAND_MAX) * 2.0 - 1.0;  // Generate values between -1 and 1
    }
}

// Make the matrix positive definite
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        if (i == j) {
            A[i][j] += n;
        } else {
            A[i][j] += ((double) rand() / RAND_MAX) * sqrt(n);
            A[j][i] = A[i][j];
        }
    }
}

for(i=0; i < n; i++) {
    for(j=0; j < n; j++) {
        L[i][j] = 0.0;
        U[i][j] = 0.0;
    }
}
end = omp_get_wtime();
printf("Initialization: %f\n", end-start);
```

Seguidament, realitzem la factorització de Cholesky per la matriu diagonal U. Per fer-ho, utilitzem la paral·lelització #pragma omp parallel. La nostra idea principal era implementar-ho abans del primer for, però l'execució no era òptima, creiem que era degut al fet que les iteracions del bucle es dividien en masses threads i en comptes d'accelerar el procés, l'alentia. Per això, hem optat per implementar-ho directament als bucles interns, en els quals si afavoria a la millora del rendiment.
En el primer bucle, utilitzem la reducció per calcular la suma total de tmp i a partir d'aquí calcular els elements diagonals. En el segon bucle, fem servir schedule(dynamic, 2) per

repartir dinàmicament el càlcul dels elements no diagonals en dos elements per threads, i intentar igualar la càrrega de càlcul de cada fil per tal d'accelerar el procés. En els dos processos, també declarem shared(U,A) per tal d'assegurar que aquestes variables són compartides per tots els threads (amb la implementació de parallel for, totes les variables són declarades com a first-private, per això no és necessari declarar les variables privades).

```
//#pragma omp parallel shared(U,A,L) private(i,j,k)
for(i=0; i<n; i++) {
    // Calculate diagonal elements
    tmp = 0.0;
    #pragma omp parallel for reduction(+:tmp) shared(U,A)
    for(k=0;k<i;k++) {
        tmp += U[k][i]*U[k][i];
    }

    U[i][i] = sqrt(A[i][i]-tmp);
    // Calculate non-diagonal elements
    #pragma omp parallel for schedule(dynamic,4) shared(U,A)
    for(j=i+1;j<n;j++) {
        tmp = 0.0;
        for(k=0;k<i;k++) {
            tmp += U[k][j]*U[k][i];
        }
        U[i][j] = (A[j][i]-tmp)/U[i][i];

        // TODO U[i][j] =
    }
}
end = omp_get_wtime();
printf("Cholesky: %f\n", end-start);
```

$$u_{ij} = \frac{a_{ji} - \sum_{k=0}^{i-1} u_{kj} u_{ki}}{u_{ii}}$$

off-diagonal elements

$$u_{ii} = \sqrt{a_{ii} - \sum_{k=0}^{i-1} u_{ki}^2}$$

diagonal elements

En la següent part del codi, calculem L transposant U. Per això, utilitzem #pragma omp parallel for collapse(2) shared(L,U) per paral·lelitzar els dos primers bucles com un únic bucle de mida n*n i repartir-ho entre els threads.

```
/**
 * 3. Calculate L from U'
 */

start = omp_get_wtime();
int strip_size = 32; // Adjust

#pragma omp parallel for collapse(2) shared(L,U)
for (k = 0; k < n; k+=strip_size) {
    for (int l = 0; l < n; l+=strip_size){
        for (i = k; i < k + strip_size && i < n; i++) {
            for (j = l; j < l + strip_size && j < n; j++) {
                L[i][j] = U[j][i];
            }
        }
    }
}

// TODO L=U'
end = omp_get_wtime();
printf("L=U': %f\n", end-start);
```

Després, calculem B=L*U. Tornem a utilitzar la reducció per calcular la suma total de tmp i seguidament calculem B per tots els valors de L i U. Afegim un omp parallel for als dos bucles exteriors.

```
#pragma omp parallel for shared(B,L,U) collapse(2)
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
        tmp = 0.0;
        #pragma omp parallel for reduction(+:tmp)
        for(int k=0; k<n; k++) {
            tmp += L[i][k]*U[k][j];
        }
        B[i][j] = tmp;
    }
}
end = omp_get_wtime();
printf("B=LU: %f\n", end-start);
```

Finalment, comprovem que les matrius A i B són iguals o molt semblants, amb una tolerància d'error de 0.001%.

```
    /**
     * 5. Check if all elements of A and B have a difference smaller than 0.001%
     */
    cnt=0;
    // TODO check if matrices are equal

    for(i=0; i<n; i++) {
        for(j=0; j<n; j++) {
            if(fabs((B[i][j]-A[i][j])/A[i][j])*100 > 0.001) {
                cnt++;
            }
        }
    }
    if(cnt != 0) {
        printf("Matrices are not equal\n");
    } else {
        printf("Matrices are equal\n");
    }
    printf("A==B?: %d\n", cnt);

    for(i=0; i<n; i++) {
        free(A[i]);
        free(L[i]);
        free(U[i]);
        free(B[i]);
    }
    free(A);
    free(L);
    free(U);
    free(B);
}
```

$$Error[\%] = |\frac{B_{ij} - A_{ij}}{A_{ij}}| \times 100$$

Al final de l'execució, obtenim els dos temps d'execució, el seqüencial i el paral·lelitzat, i obtenim els diferents temps:

PART2: En el scheduler, escollim el dynamic de mida quatre perquè és el que triga menys temps en executar-se. Per comprovar-ho, hem provat amb diverses chunks sizes.

schedule	*chunk size*	*temps d'execució*

static	*1*	*Cholseky: 14.367564*
		`OpenMP Cholesky`
		`Initialization: 0.245625`
		`Cholesky: 14.367564`
		`L=U': 0.035074`
		`B=LU: 54.657695`
		`Matrices are equal`
		`A==B?: 0`

static	*2*	*Cholesky: 7.188666* OpenMP Cholesky Initialization: 0.227143 Cholesky: 7.188666 L=U': 0.020133 B=LU: 43.345013 Matrices are equal A==B?: 0
dynamic	*1*	*Cholesky: 9.160827* OpenMP Cholesky Initialization: 0.248451 Cholesky: 9.160827 L=U': 0.019931 B=LU: 56.553265 Matrices are equal A==B?: 0
dynamic	*2*	*Cholesky: 8.249746* OpenMP Cholesky Initialization: 0.259745 Cholesky: 8.249746 L=U': 0.026921 B=LU: 54.069939 Matrices are equal A==B?: 0
dynamic	*4*	*Cholesky: 4.649036* OpenMP Cholesky Initialization: 0.233917 Cholesky: 4.649036 L=U': 0.015190 B=LU: 40.521409 Matrices are equal A==B?: 0
dynamic	*10*	*Cholesky: 5.554645* OpenMP Cholesky Initialization: 0.228663 Cholesky: 5.554645 L=U': 0.032063 B=LU: 53.716142 Matrices are equal A==B?: 0
guided	*2*	*Cholesky: 5.162076* OpenMP Cholesky Initialization: 0.239160 Cholesky: 5.162076 L=U': 0.031915 B=LU: 54.306520 Matrices are equal A==B?: 0

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

Amb n=3000, hem executat Cholseky i la multiplicació de matrius per als diferents cores:

1 core

```
OpenMP Cholesky
Initialization: 0.237862
Cholesky: 18.029199
L=U': 0.031234
B=LU: 92.237770
Matrices are equal
A==B?: 0
```

2 cores

```
OpenMP Cholesky
Initialization: 0.240838
Cholesky: 13.013422
L=U': 0.043902
B=LU: 57.071458
Matrices are equal
A==B?: 0
```

4 cores

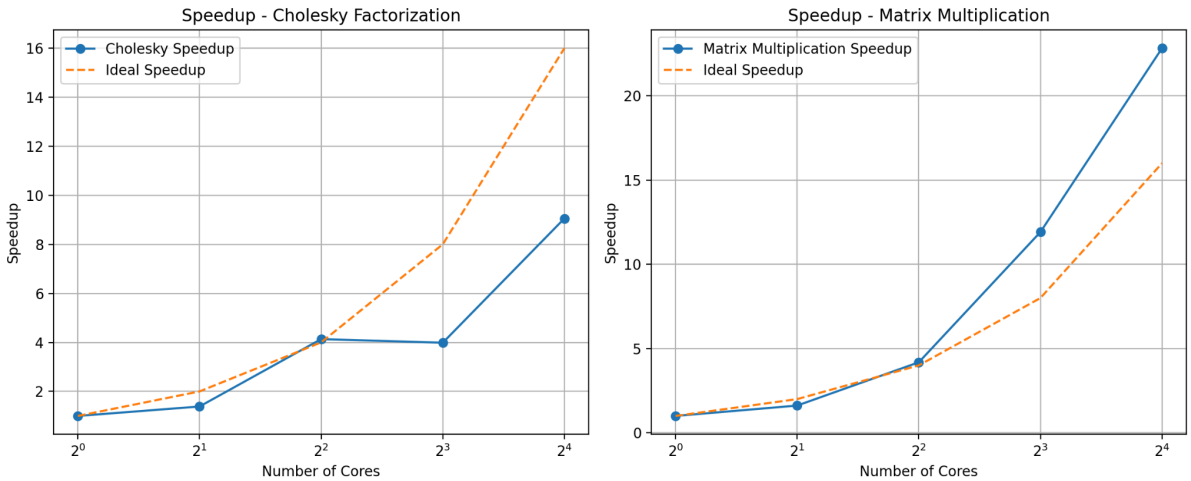
```
OpenMP Cholesky
Initialization: 0.212863
Cholesky: 4.367848
L=U': 0.028952
B=LU: 22.004131
Matrices are equal
A==B?: 0
```

8 cores

```
OpenMP Cholesky
Initialization: 0.170010
Cholesky: 4.527913
L=U': 0.070757
B=LU: 7.740717
Matrices are equal
A==B?: 0
```

16 cores

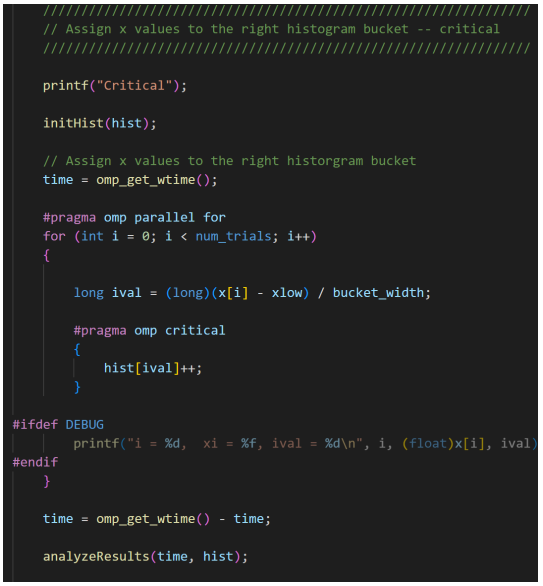
```
OpenMP Cholesky
Initialization: 0.167789
Cholesky: 1.998615
L=U': 0.041472
B=LU: 4.046138
Matrices are equal
A==B?: 0
```



Podem veure que en els dos casos, el nostre speed up s'assembla bastant al cas a l'ideal, exceptuant alguns valors puntuals. Per tant, podem concloure que el nostre algoritme té un temps d'execució adequat i escala bé a mesura que augmentem el nombre de cores.

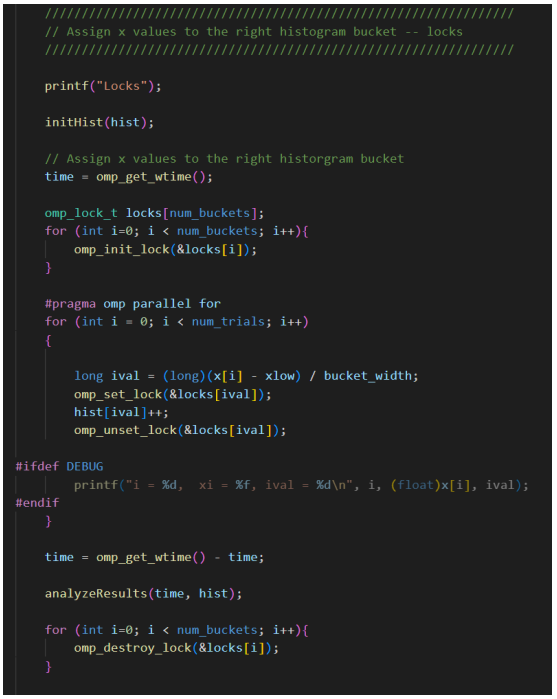
2. Histogram

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

Mètode de paral·lelització	**Codi**
Critical: Només un thread pot accedir i modificar hist[ival] a la vegada. Fàcil d'implementar però molt lent.	
Atomic: Llegeix i actualitza el valor de hist[ival]. Protegeix la suma. Més eficient que el critical, especialment en aquest cas que s'ha de realitzar una operació simple.	

```
    ////////////////////////////////////////////////////////////
    // Assign x values to the right histogram bucket -- critical
    ////////////////////////////////////////////////////////////

    printf("Critical");

    initHist(hist);

    // Assign x values to the right historgram bucket
    time = omp_get_wtime();

    #pragma omp parallel for
    for (int i = 0; i < num_trials; i++)
    {

        long ival = (long)(x[i] - xlow) / bucket_width;

        #pragma omp critical
        {
            hist[ival]++;
        }

#ifdef DEBUG
        printf("i = %d,  xi = %f, ival = %d\n", i, (float)x[i], ival);
#endif
    }

    time = omp_get_wtime() - time;

    analyzeResults(time, hist);
```

```
    ////////////////////////////////////////////////////////////
    // Assign x values to the right histogram bucket -- atomic
    ////////////////////////////////////////////////////////////

    printf("Atomic");

    initHist(hist);

    // Assign x values to the right historgram bucket
    time = omp_get_wtime();

    #pragma omp parallel for
    for (int i = 0; i < num_trials; i++)
    {

        long ival = (long)(x[i] - xlow) / bucket_width;

        #pragma omp atomic
        {
            hist[ival]++;
        }

#ifdef DEBUG
        printf("i = %d,  xi = %f, ival = %d\n", i, (float)x[i], ival);
#endif
    }

    time = omp_get_wtime() - time;

    analyzeResults(time, hist);
```

Locks: Crea un lock per cada block del histograma i si s'ha d'augmentar hist[ival], es fa lock i unlock del lock corresponent al bucket desitjat.	
Reduction: Cada thread fa la seva suma de hist[ival] en la seva copia de l'array i al final del bucle se sumen tots els valors resultants de cada thread per a obtenir el resultat final de cada bucket.	

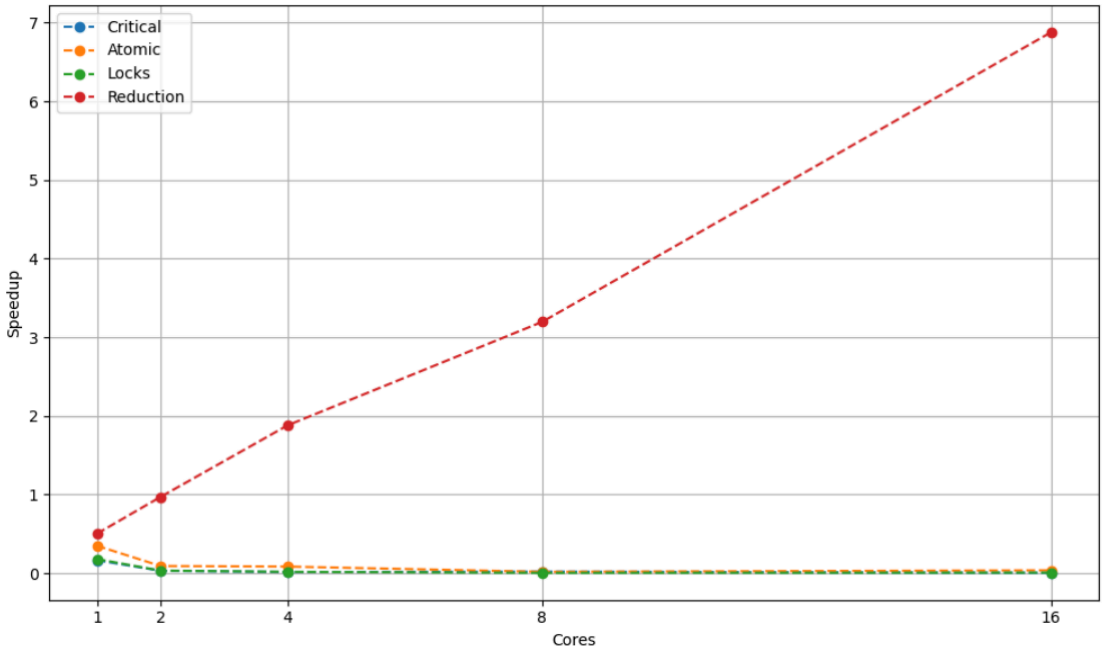
```
    ////////////////////////////////////////////////////////////
    // Assign x values to the right histogram bucket -- locks
    ////////////////////////////////////////////////////////////

    printf("Locks");

    initHist(hist);

    // Assign x values to the right historgram bucket
    time = omp_get_wtime();

    omp_lock_t locks[num_buckets];
    for (int i=0; i < num_buckets; i++){
        omp_init_lock(&locks[i]);
    }

    #pragma omp parallel for
    for (int i = 0; i < num_trials; i++)
    {

        long ival = (long)(x[i] - xlow) / bucket_width;
        omp_set_lock(&locks[ival]);
        hist[ival]++;
        omp_unset_lock(&locks[ival]);

#ifdef DEBUG
        printf("i = %d,  xi = %f, ival = %d\n", i, (float)x[i], ival);
#endif
    }

    time = omp_get_wtime() - time;

    analyzeResults(time, hist);

    for (int i=0; i < num_buckets; i++){
        omp_destroy_lock(&locks[i]);
    }
```

```
    ////////////////////////////////////////////////////////////
    // Assign x values to the right histogram bucket -- reduction
    ////////////////////////////////////////////////////////////

    printf("Reduction");

    initHist(hist);

    // Assign x values to the right historgram bucket
    time = omp_get_wtime();

    #pragma omp parallel for reduction(+:hist[:num_buckets])
    for (int i = 0; i < num_trials; i++)
    {

        long ival = (long)(x[i] - xlow) / bucket_width;

        hist[ival]++;

#ifdef DEBUG
        printf("i = %d,  xi = %f, ival = %d\n", i, (float)x[i], ival);
#endif
    }

    time = omp_get_wtime() - time;

    analyzeResults(time, hist);


    free(x);
    return 0;
}
```

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

Les diferències de temps entre els mètodes paral·lels es deuen a la sincronització dels threads per cada mètode. El mètode critical és el més lent, ja que només permet que un fil actualitzi l'histograma a la vegada, i pot generar coll d'ampolla. L'atomic millora el rendiment sincronitzant només operacions simples, i és més eficient que el critical. Per altra banda, l'ús de locks ofereix un millor rendiment que critical i atomic, perquè s'associa un bloqueig per cada bucket, permetent una major concurrència. Finalment, la versió amb reduction és la més eficient, ja que cada thread acumula els seus propis resultats i al final es combinen, eliminant la necessitat de sincronització durant l'execució principal.

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



Podem veure que l'únic mètode que escala bé és la reducció. A mesura que augmentem el nombre de cores el speedup també i, per tant, el temps d'execució en paral·lel disminueix en comparació al seqüencial. En canvi, per la resta de mètodes, la creació de més threads produeix una sobrecàrrega i fa que el speedup no millori. En el cas de critical en afegir més threads fem que l'actualització de l'histograma sigui lenta, ja que només pot entrar a la regió crítica un thread. En atomic evitem això, però igualment hi ha sobrecàrrega de threads i no s'utilitzen eficientment. En locks la creació de locks i la seva destrucció comporta massa temps. L'opció de la reducció és la correcta, perquè fa una còpia de l'array per cada thread i cadascun l'actualitza independentment, fent que cap thread s'hagi d'esperar. Finalment se sumen els resultats de cada array.

1 core

```
1 threads
Sequential histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001047 seconds
Critical histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.006378 seconds
Atomic histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.003005 seconds
Locks histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.005858 seconds
Reduction histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.002091 seconds
```

2 cores

```
2 threads
Sequential histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001015 seconds
Critical histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.033384 seconds
Atomic histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.011206 seconds
Locks histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.031652 seconds
Reduction histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001047 seconds
```

4 cores

```
4 threads
Sequential histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001004 seconds
Critical histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.075039 seconds
Atomic histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.012016 seconds
Locks histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.059301 seconds
Reduction histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.000534 seconds
```

8 cores

```
8 threads
Sequential histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001016 seconds
Critical histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.052609 seconds
Atomic histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.062060 seconds
Locks histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.314673 seconds
Reduction histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.000318 seconds
```

16 cores

```
16 threads
Sequential histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.001004 seconds
Critical histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.226105 seconds
Atomic histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.028294 seconds
Locks histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.167522 seconds
Reduction histogram for 50 buckets of 1000000 values
 ave = 20000.000000, std_dev = 394.372925
 in 0.000146 seconds
```

3. Argmax

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

```c
void argmax_seq(double *v, int N, double *m, int *idx_m) {
    *m = v[0];
    *idx_m = 0;
    for (int i = 1; i < N; i++) {
        if (v[i] > *m) {
            *m = v[i];
            *idx_m = i;
        }
    }
}
```

Per fer la implementació seqüencial hem inicialitzat les variables *m que guarda el màxim en l'array i *idx_m que guarda l'índex del valor màxim amb els valors del primer número de l'array (per defecte). Fem un bucle for per iterar sobretot l'array i trobar el màxim, actualitzant l'índex.

```c
// computes the argmax in parallel with a for loop
void argmax_par(double *v, int N, double *m, int *idx_m) {
    *m = v[0];
    *idx_m = 0;

    #pragma omp parallel
    {
        double thread_m = v[0];
        int thread_idx = 0;

        #pragma omp for
        for (int i = 1; i < N; i++) {
            if (v[i] > thread_m) {
                thread_m = v[i];
                thread_idx = i;
            }
        }

        #pragma omp critical
        {
            if (thread_m > *m) {
                *m = thread_m;
                *idx_m = thread_idx;
            }
        }
    }
}
```

En la implementació paral·lela de argmax afegim una regió paral·lela per fer el bucle for i l'actualització del valor màxim i el seu índex. Primer vam intentar fer-ho a partir d'una reducció amb max i que cada thread trobés el seu màxim i després combinar-los per trobar el global, però ens vam trobar amb la dificultat de l'actualització de l'índex. Per això, hem decidit crear una variable local per a cada thread on guarda l'índex i valor màxim de la part d'array que està analitzant (semblant a una reducció). En acabat, actualitzem el valor de *m i *idx_m si el valor que cada thread té com a màxim és més gran que el global. Aquesta part s'ha de protegir, ja que les variables a actualitzar són compartides entre threads. Hem optat per utilitzar critical, per tal que només un thread a la vegada pugui entrar a la regió i actualitzar els valors.

```c
// computes the argmax recursively and sequentially
void argmax_recursive(double *v, int N, double *m, int *idx_m, int K) {
    if(N<K){
        argmax_seq(v,N,m,idx_m);
        return;
    }
    else{
        double max1;
        double max2;
        int idx_1;
        int idx_2;
        int mid = N/2;

        argmax_recursive(v, mid, &max1, &idx_1, K);
        argmax_recursive(v + mid, N - mid, &max2, &idx_2, K);

        if(max1>max2){
            *m = max1;
            *idx_m = idx_1;
        }else{
            *m = max2;
            *idx_m = idx_2 + mid;
        }
    }
}
```

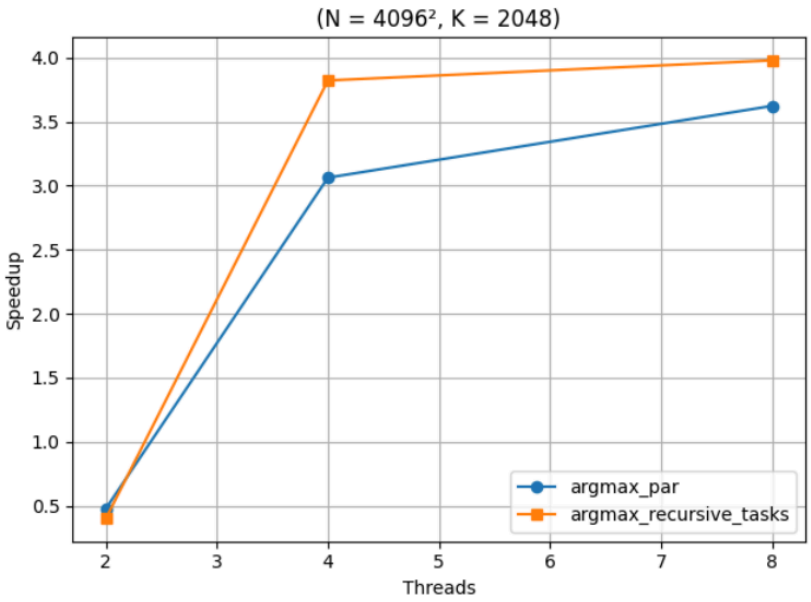
En la implementació recursiva de argmax hem definit el cas base tal com ens demana l'enunciat, que ens diu que quan la mida de l'array a analitzar és menor a K elements ha de calcular el màxim i el seu índex cridant a argmax_seq. En cas contrari, hem definit 4 variables locals per guardar el màxim i el seu índex en les dues meitats en què dividim l'array i poder després calcular el màxim global del array. Tal com hem mencionat, per tal de calcular el màxim recursivament, dividim l'array per la meitat tantes vegades fins que sigui de la mida de K. Un cop arribem al cas base la funció argmax_seq retorna el màxim d'aquell array. Per tal de calcular el màxim del array de mida N original, combinem els

resultats de les dues meitats. Per fer això mirem quin dels dos valors màxims de cada meitat de l'array és el màxim global i actualitzem.

```
void argmax_recursive_tasks(double *v, int N, double *m, int *idx_m, int K) {
    if(N<=K){
        argmax_seq(v,N,m,idx_m);
        return;
    }
    else{
        double max1;
        double max2;
        int idx_1;
        int idx_2;
        int mid = N/2;

        #pragma omp task shared(max1,idx_1)
        argmax_recursive(v,mid,&max1,&idx_1,K);
        #pragma omp task shared(max2,idx_2)
        argmax_recursive(v+mid,N-mid,&max2,&idx_2,K);

        #pragma omp taskwait
        if(max1>max2){
            *m = max1;
            *idx_m = idx_1;
        }else{
            *m = max2;
            *idx_m = idx_2 + mid;
        }
    }
}
```

Per a la implementació paral·lela recursiva, fem servir tasks, ja que divideix el codi en parts independents que pot executar qualsevol thread disponible. Les tasks del nostre programa són les crides recursives, perquè aquesta part del codi el pot executar qualsevol thread independentment de l'altra task. No fa falta cap control de variables compartides, pel fet que cada thread modifica només les variables que comparteix la task, que són el max1, max2 i idx_1, idx_2. El que sí que és important és posar un taskwait a l'hora de combinar els màxims dels dos arrays. Ens hem d'esperar que les dues tasques acabin perquè els màxims i índex estiguin calculats. Creem la regió paral·lela quan cridem la funció recursiva i fem que només un thread executi la funció per tal que es creï una task cada vegada que cridem recursivament la funció i no varies. Això no obstant, com estem en una regió paral·lela, qualsevol thread pot executar la task.

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.



Podem veure que el speedup del codi en paral·lel creix a mesura que augmentem els threads. No creix linealment i s'estanca a mesura que afegim més threads, a causa de la sobrecàrrega de crear i destruir els threads, dels diferents treballs en el node i també pot haver-hi coll d'ampolla. En el codi recursiu escala millor, ja que pot dividir la feina més equilibradament. Així i tot, necessitaríem moltes més mesures de temps de cada codi per cada thread (hem agafat la que dona més speedup) i que no hi hagués soroll d'altres treballs.

2 threads

```
Running argmax with K = 2048 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.019099
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.022772
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.009331
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.027175
```

4 threads

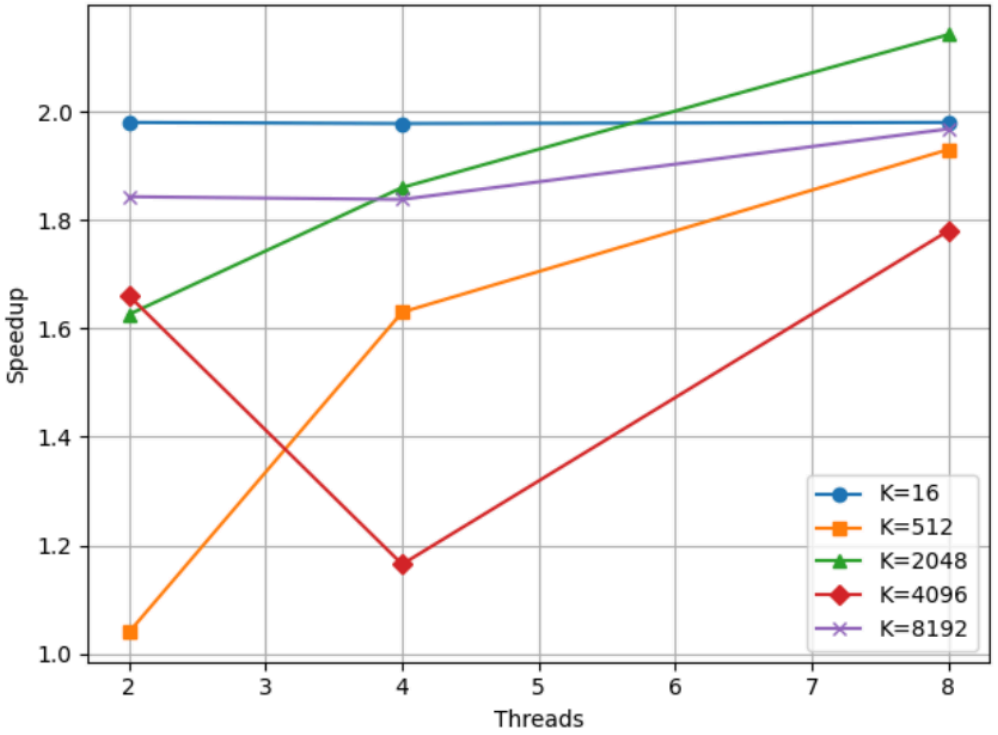
```
Running argmax with K = 2048 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014844
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.006234
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010783
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.004995
```

8 threads

```
Running argmax with K = 2048 using 8 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.013716
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.005270
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.009344
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.004799
```

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.



El gràfic no és precís, en treballar amb temps tan petits, qualsevol canvi ni que sigui mínim altera molt el speedup. A més a més, com que hi ha altres treballs en el sistema els temps van canviant amb poca coherència.

Tot i això, podem observar que en K = 16 el array en què busquem el mínim és massa petit, pel fet que hem d'estar llegint constantment parts de l'array en la memòria i calcular el màxim i podem veure que el speedup no creix.

En K = 512 i K = 2048 el speedup escala millor i podem veure com a mesura que augmentem els processos el speedup també.

En K = 4096 hi ha una caiguda del speedup en 4 threads que es pot donar per soroll extern. En K = 8192 comença amb bon speedup, però es manté a mesura que augmentem els processos.

K = 16

```
Running argmax with K = 16 using 1 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015331
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.015265
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.014509
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.014162
```

```
Running argmax with K = 16 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014959
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.008137
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.014796
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.007320
```

```
Running argmax with K = 16 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015638
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.004367
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.013868
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.007333
```

```
Running argmax with K = 16 using 8 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015492
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.007968
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.014163
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.007323
```

K = 512

```
Running argmax with K = 512 using 1 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014423
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.014325
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.011343
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.013867
```

```
Running argmax with K = 512 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015221
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.007700
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010705
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005551
```

```
Running argmax with K = 512 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.013788
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.005351
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010315
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.006948
```

```
Running argmax with K = 512 using 8 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014869
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.004554
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010459
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005869
```

K = 2048

```
Running argmax with K = 2048 using 1 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015564
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.015384
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010604
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010679
```

```
Running argmax with K = 2048 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014854
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.007649
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010622
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.006521
```

```
Running argmax with K = 2048 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015176
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.004004
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010581
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005696
```

```
Running argmax with K = 2048 using 8 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.013742
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.005767
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.009485
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.004951
```

K = 4096

```
Running argmax with K = 4096 using 1 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.013706
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.013728
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.009578
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.009367
```

```
Running argmax with K = 4096 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015487
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.007875
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.011694
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005785
```

```
Running argmax with K = 4096 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.014558
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.005682
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010996
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.008222
```

K = 8192

```
Running argmax with K = 8192 using 1 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015334
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.015003
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010341
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010993
```

```
Running argmax with K = 8192 using 2 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015479
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.007991
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010700
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005610
```

```
Running argmax with K = 8192 using 4 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015411
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.004436
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010627
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005626
```

```
Running argmax with K = 8192 using 8 threads
Sequential for argmax: m = 1.000000, idx_m = 8388608, time = 0.015220
Parallel for argmax: m = 1.000000, idx_m = 8388608, time = 0.003971
Sequential recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.010294
Parallel recursive argmax: m = 1.000000, idx_m = 8388608, time = 0.005254
```

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?

La intensitat aritmètica es defineix com a q = N/V on N és la quantitat de treball que hem de fer (flops) i V la quantitat de dades que necessitem de la memòria (bytes). Per calcular el màxim i índex de l'array de mida N necessitem una comparació aproximadament per cada posició de l'array i portem de memòria N*8 bytes (double), per tant, q = N/N*8 = 1/8. La intensitat aritmètica és molt baixa, ja que realitzem 1/8 de flops per cada byte, fem molt poques operacions per la quantitat de dades que portem de memòria.

El recurs que creiem que és el coll d'ampolla és la memory bandwidth, perquè per calcular el màxim necessitem carregar l'array de la memòria. En el cas recursiu, si posem un valor de K molt baix estarem portant de memòria parts de l'array constantment. La capacitat d'operació no pot ser el coll d'ampolla, perquè fem poques operacions en comparació amb les dades que necessitem.