Marc Mezquita Meseguer 1455119
Oriol Martínez Acón 1460561

# OPENMP

## Plantejament previ a optimitzar

Per a començar vam provar de crear la regió paral·lela dins del bucle “epoch” i més concretament dins de la regió de cada “batch”. Al fer-ho ens vam donar compte que això generava molt “overhead” i que era més rentable definir una regió paral·lela fora del bucle de “epoch”. Encara que fora del bucle es repeteixen les iteracions per a cada thread, acaba sent menys costós que crear, sincronitzar i destruir aquests.

## Optimitzacions

***Primer canvi al codi***

```
for( int epoch = 0 ; epoch < 1000000 ; epoch++) {    // iterate weight updates
  for( int p = 0 ; p < NUMPAT ; p++ )   // randomize order of individuals
        ranpat[p] = p ;
  for( int p = 0 ; p < NUMPAT ; p++) {
        int x = rando();
        int np = (x*x)%NUMPAT;
        int op = ranpat[p] ; ranpat[p] = ranpat[np] ; ranpat[np] = op ;
  }
  Error = BError = 0.0;

  printf("."); fflush(stdout);
```

*Imatge Original*

```
#pragma omp parallel
{

  for( int epoch = 0 ; epoch < 1000000 ; epoch++) {    // iterate weight updates

    #pragma omp for
    for( int p = 0 ; p < NUMPAT ; p++ )   // randomize order of individuals
      {ranpat[p] = p ;}

    #pragma omp single
    {
      for( int p = 0 ; p < NUMPAT ; p++) {
        int x = rando();
        int np = (x*x)%NUMPAT;
        int op = ranpat[p] ; ranpat[p] = ranpat[np] ; ranpat[np] = op ;
      }

      Error = BError = 0.0;

      printf("."); fflush(stdout);
    }
```

*Imatge del codi modificat*

En aquest cas, com hem comentat anteriorment, vàrem definir la regió paralela abans del “epoch”, un cop fet vam anar bucle a bucle comprovant si era paral·lelitzable, i si no ho era, que podìem fer per que això canviï.

El primer bucle que ens vàrem trobar va ser guardat a ranpat[p] el valor de p, això és completament paral·lelitzable llavors simplement vam utilitzar la directiva #pragma omp for.

El següent bucle no es paral·lelitzable perquè hi han dependencies entre cada iteració, en el cas de que volguèssim paralelitzar hauriem de esbrinar una manera de fer l'intercanvi de posicions a l'array sense que hi hagi la possibilitat de duplicar dades.

***Segon canvi al codi***

```
for( int j = 0 ; j < NUMHID ; j++ ) {    // compute hidden unit activations
    double SumH = 0.0;
    for( int i = 0 ; i < NUMIN ; i++ ) SumH += tSet[p][i] * WeightIH[j][i];
    Hidden[p][j] = 1.0/(1.0 + exp(-SumH)) ;
}
for( int k = 0 ; k < NUMOUT ; k++ ) {    // compute output unit activations and error
    double SumO = 0.0;
    for( int j = 0 ; j < NUMHID ; j++ ) SumO += Hidden[p][j] * WeightHO[k][j] ;
    Output[p][k] = 1.0/(1.0 + exp(-SumO)) ;   // Sigmoidal Outputs
    BError += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;
    DeltaO[k] = (Target[p][k] - Output[p][k]) * Output[p][k] * (1.0 - Output[p][k]) ;
}
```

*Imatge Original*

```
//for 1
#pragma omp for
for( int j = 0 ; j < NUMHID ; j++ ) {    // compute hidden unit activations
  double SumH = 0.0;
  for( int i = 0 ; i < NUMIN ; i++ ) SumH += tSet[p][i] * WeightIH[j][i]; //*** UTI
  Hidden[p][j] = 1.0/(1.0 + exp(-SumH)) ;
}

//for 2
#pragma omp for reduction(+:BError)
for( int k = 0 ; k < NUMOUT ; k++ ) {    // compute output unit activations and err
  double SumO = 0.0;
  for( int j = 0 ; j < NUMHID ; j++ ) SumO += Hidden[p][j] * WeightHO[k][j] ; //***
  Output[p][k] = 1.0/(1.0 + exp(-SumO)) ;   // Sigmoidal Outputs
  //*** UTILITZAR PATRON REDUCE
  BError += 0.5 * (Target[p][k] - Output[p][k]) * (Target[p][k] - Output[p][k]) ;
  DeltaO[k] = (Target[p][k] - Output[p][k]) * Output[p][k] * (1.0 - Output[p][k]) ;
}
```

*Imatge del codi modificat*

Com podem veure en el primer bucle és totalment paral·lelitzable pels thread degut a que no hi han dependències entre iteracions, per aquesta mateixa raó utilitzarem la directiva "#pragma omp for".

En el segon bucle tenim un problema amb les dependències. El BError és una variable compartida pels threads i per la qual cosa no es pot paral·lelitzar amb la mateixa directiva que en el cas anterior. La única forma de poder paral·lelitzar totalment aquest bucle és a través de la directiva reduction. Aquesta directiva ens permet crear variables BError locals per a cada thread. Un cop acabades les iteracions aquestes variables es sumaràn en una única operació a la variable global BError, d'aquesta forma evitem el problema de la memòria compartida.

***Tercer canvi al codi***

```
for( int j = 0 ; j < NUMHID ; j++ ) {     // update delta weights DeltaWeightIH
    double SumDOW = 0.0 ;
    for( int k = 0 ; k < NUMOUT ; k++ ) SumDOW += WeightHO[k][j] * DeltaO[k] ;
    DeltaH[j] = SumDOW * Hidden[p][j] * (1.0 - Hidden[p][j]) ;
    for( int i = 0 ; i < NUMIN ; i++ )
        DeltaWeightIH[j][i] = eta * tSet[p][i] * DeltaH[j] + alpha * DeltaWeightIH[j][i];
}
```

*Imatge Original*

```
//for 3
#pragma omp for
for( int j = 0 ; j < NUMHID ; j++ ) {     // update delta weights DeltaWeightIH
  double SumDOW = 0.0 ;
  for( int k = 0 ; k < NUMOUT ; k++ ) SumDOW += WeightHO[k][j] * DeltaO[k] ; //*** U
  DeltaH[j] = SumDOW * Hidden[p][j] * (1.0 - Hidden[p][j]) ;
}

//new for
#pragma omp for collapse(2)
for( int j = 0 ; j < NUMHID ; j++ ) {     // update delta weights DeltaWeightIH
  for( int i = 0 ; i < NUMIN ; i++ )
    DeltaWeightIH[j][i] = eta * tSet[p][i] * DeltaH[j] + alpha * DeltaWeightIH[j][i];
}
```

*Imatge del codi modificat*

En aquest cas vàrem decidir dividir el bucle principal en dos bucles, això ho vàrem fer perquè el bucle mes intern cada thread l'havia d'executar molts cops i això genera un "overhead" molt elevat al tenir que procesar molta informació cadascun d'ells. Llavors vàrem dividir el bucle i al separar aquestes últimes iteracions la feina era més lleugera amb la directiva #pragma omp for collapse(2), la cual fusiona els dos bucles com si fossin un sol.

***Quart canvi al codi***

```
    for( int k = 0 ; k < NUMOUT ; k ++ )     // update delta weights DeltaWeightHO
        for( int j = 0 ; j < NUMHID ; j++ )
            DeltaWeightHO[k][j] = eta * Hidden[p][j] * DeltaO[k] + alpha * DeltaWeightHO[k][j];
}
```

*Imatge Original*

```
    #pragma omp for collapse(2)
    for(int k = 0 ; k < NUMOUT ; k ++ )     // update delta weights DeltaWeightHO
      for(int j = 0 ; j < NUMHID ; j++ )
        DeltaWeightHO[k][j] = eta * Hidden[p][j] * DeltaO[k] + alpha * DeltaWeightHO[k][j];

}
```

*Imatge del codi modificat*

Aquest bucle al ser completament paral·lelitzable i ser dos bucles niats vam utilitzar la directiva #pragma omp for collapse(2).

***Cinquè canvi al codi***

```
Error += BError;
           for( int j = 0 ; j < NUMHID ; j++ )     // update weights WeightIH
                   for( int i = 0 ; i < NUMIN ; i++ )
                       WeightIH[j][i] += DeltaWeightIH[j][i] ;

           for( int k = 0 ; k < NUMOUT ; k ++ )    // update weights WeightHO
                   for( int j = 0 ; j < NUMHID ; j++ )
                       WeightHO[k][j] += DeltaWeightHO[k][j] ;
```

*Imatge Original*

```
    #pragma omp single
    {
      Error += BError;
    }

    //int i,j,k;
    #pragma omp for collapse(2)
    for(int j = 0 ; j < NUMHID ; j++ )      // update weights WeightIH
      for(int i = 0 ; i < NUMIN ; i++ )
        WeightIH[j][i] += DeltaWeightIH[j][i] ;

    #pragma omp for collapse(2)
    for(int k = 0 ; k < NUMOUT ; k ++ )     // update weights WeightHO
      for(int j = 0 ; j < NUMHID ; j++ )
        WeightHO[k][j] += DeltaWeightHO[k][j] ;
```

*Imatge del codi modificat*

En aquest cas els dos bucles no tenen cap dependència entre les iteracions i per la qual cosa son perfectament paral·lelitzables, a més a més, al ser bucles niats podem aplicar la directiva “collapse” que ens permet unificar-los com si fos un únic bucle.

***Sisè canvi al codi***

```
  Error = Error/((NUMPAT/BSIZE)*BSIZE); //mean error for the last epoch
  if( !(epoch%100) ) printf("\nEpoch %-5d :   Error = %f \n", epoch, Error) ;
  if( Error < 0.0004 ) {
  printf("\nEpoch %-5d :   Error = %f \n", epoch, Error) ; break ;  // stop le
  }
```

*Imatge Original*

```
  #pragma omp single
  {
    Error = Error/((NUMPAT/BSIZE)*BSIZE); //mean error for the last epoch
    if( !(epoch%100) ) printf("\nEpoch %-5d :   Error = %f \n", epoch, Error) ;
  }

  if( Error < 0.0004 ) {
      printf("\nEpoch %-5d :   Error = %f \n", epoch, Error) ; break ;  // stop
  }
#pragma omp barrier
```

*Imatge del codi modificat*

En aquest cas hem declarat amb la directiva single la primera assignació de la variable Error (variable compartida), degut a que no volem que cada thread re-calculi el valor d’aquesta i finalment el primer “if” per a mostrar un cop només per cada 100 “epoch” el print.

***Setè canvi al codi***

```
for( int p = 0 ; p < NUMRPAT ; p++ ) {    // repeat for all the recognition pattern
    for( int j = 0 ; j < NUMHID ; j++ ) {    // compute hidden unit activations
          double SumH = 0.0;
          for( int i = 0 ; i < NUMIN ; i++ ) SumH += rSet[p][i] * WeightIH[j][i];
          Hidden[j] = 1.0/(1.0 + exp(-SumH)) ;
    }

    for( int k = 0 ; k < NUMOUT ; k++ ) {    // compute output unit activations
          double SumO = 0.0;
          for( int j = 0 ; j < NUMHID ; j++ ) SumO += Hidden[j] * WeightHO[k][j] ;
          Output[k] = 1.0/(1.0 + exp(-SumO)) ;   // Sigmoidal Outputs
    }
    printRecognized(p, Output);
}
```

*Imatge Original*

```
#pragma omp parallel
{
  for( int p = 0 ; p < NUMRPAT ; p++ ) {    // repeat for all the recognition pat
    #pragma omp for
    for( int j = 0 ; j < NUMHID ; j++ ) {    // compute hidden unit activations
        double SumH = 0.0;
        for( int i = 0 ; i < NUMIN ; i++ ) SumH += rSet[p][i] * WeightIH[j][i]; /
        Hidden[j] = 1.0/(1.0 + exp(-SumH)) ;
    }
    #pragma omp for
    for( int k = 0 ; k < NUMOUT ; k++ ) {    // compute output unit activations
        double SumO = 0.0;
        for( int j = 0 ; j < NUMHID ; j++ ) SumO += Hidden[j] * WeightHO[k][j] ;
        Output[k] = 1.0/(1.0 + exp(-SumO)) ;   // Sigmoidal Outputs
    }
    #pragma omp single
    {
      printRecognized(p, Output);
    }
  }
```

*Imatge del codi modificat*

Per finalitzar vàrem fer canvis a la funció run() que encara que sabem que consumeix molt poc temps de l'execució vam decidir fer alguns canvis.

Van generar la regió paralela just abans de realitzar els bucles de la funció i aquestos bucles eren completament paral·lelitzables llavors simplement vam utilitzar la directiva #pragma omp for. En canvi el “print” no volíem que ho fessin tots els threads, llavors ho vam protegir amb #pragma omp single perque nomes es fes un cop.

## Speedup i compilacions

El nostre codi el vàrem compilar amb un arxiu .sub, el que feiem era posar-ho en cola per a que el WILMA l’executi.

*Imatge de l'arxiu .sub*

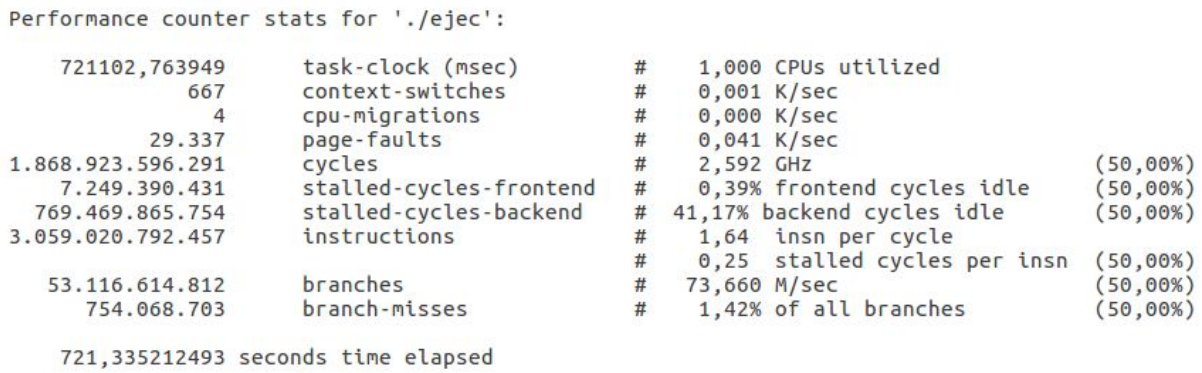
```
#! /bin/bash -l
#SBATCH -N 1
#SBATCH -n 2
#SBATCH --exclusive

export OMP_NUM_THREADS=12
gcc -Ofast common.c nn-vo-db.c -o ejec -std=c99 -lm -fopenmp
perf stat ./ejec
```

D’aquesta forma especifiquem que el nostre programa s’executi de forma exclusiva sense que algun altre programa s’executi al mateix temps. També es concreta el nombre de threads, que nosaltres hem especificat 12.

La compilació la fem mitjançant la comanda gcc amb -Ofast i -fopenmp per a les directives OMP.

*Imatge del rendiment original (single-thread)*

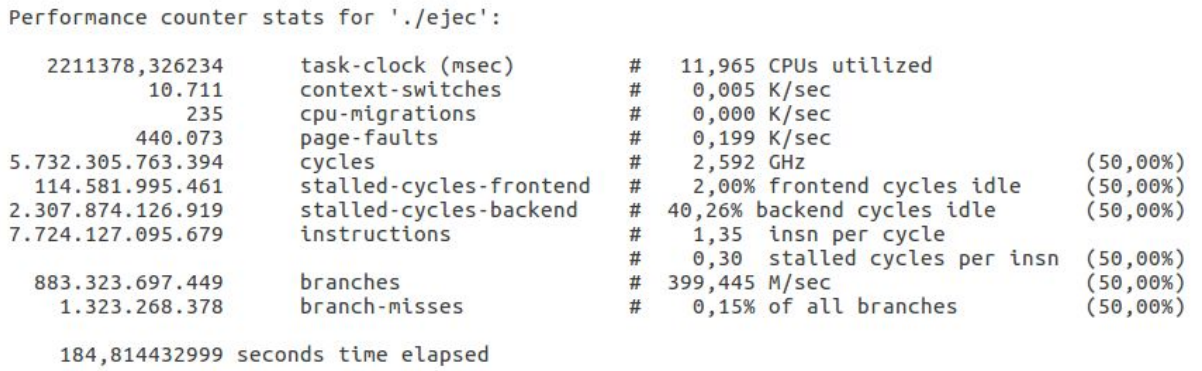


```
Performance counter stats for './ejec':

    721102,763949      task-clock (msec)         #    1,000 CPUs utilized
              667      context-switches          #    0,001 K/sec
                4      cpu-migrations            #    0,000 K/sec
           29.337      page-faults               #    0,041 K/sec
1.868.923.596.291      cycles                    #    2,592 GHz                      (50,00%)
    7.249.390.431      stalled-cycles-frontend   #    0,39% frontend cycles idle     (50,00%)
  769.469.865.754      stalled-cycles-backend    #   41,17% backend cycles idle      (50,00%)
3.059.020.792.457      instructions              #    1,64  insn per cycle
                                                 #    0,25  stalled cycles per insn  (50,00%)
   53.116.614.812      branches                  #   73,660 M/sec                    (50,00%)
      754.068.703      branch-misses             #    1,42% of all branches          (50,00%)

    721,335212493 seconds time elapsed
```

*Imatge del rendiment optimitzat (multi-thread)*



```
Performance counter stats for './ejec':

   2211378,326234      task-clock (msec)         #   11,965 CPUs utilized
          10.711      context-switches          #    0,005 K/sec
             235      cpu-migrations            #    0,000 K/sec
         440.073      page-faults               #    0,199 K/sec
5.732.305.763.394      cycles                    #    2,592 GHz                      (50,00%)
  114.581.995.461      stalled-cycles-frontend   #    2,00% frontend cycles idle     (50,00%)
2.307.874.126.919      stalled-cycles-backend    #   40,26% backend cycles idle      (50,00%)
7.724.127.095.679      instructions              #    1,35  insn per cycle
                                                 #    0,30  stalled cycles per insn  (50,00%)
  883.323.697.449      branches                  #  399,445 M/sec                    (50,00%)
    1.323.268.378      branch-misses             #    0,15% of all branches          (50,00%)

    184,814432999 seconds time elapsed
```

Com podem observar a les imatges, el temps de l'execució original es de 721 segons i el temps de l’execució optimitzada es 184 segons.
Per a calcular l’speed-up dividim 721/184 i ens dona un speedup de 3,9.