Exam June 2022

Given the following SLURM script, how many different jobs are going to be executed? How many files are going to be created in total? What are their names and contents?

```
#!/bin/sh

#SBATCH --job-name=mega

#SBATCH --nodes=1

#SBATCH --ntasks=1

#SBATCH --array=1-5

run1=$((100 + ${SLURM1002042_ARRAY_TASK_ID}))

run2=$((200 + ${SLURM_ARRAY_TASK_ID}))

run1_name=${SLURM_JOB_NAME}_${SLURM_ARRAY_TASK_ID}_1

run2_name=${SLURM_JOB_NAME}_${SLURM_ARRAY_TASK_ID}_2

echo $run1 > ${run1_name}.dat

echo $run2 > ${run2_name}.dat
```

There is an error and SLURM1002042=SLURM.

5 jobs are executed and we should obtain 15 files:

- 5 slurm
- 10 mega → run1 and run2 variables

2.- (1 point)
A program written in OpenMPI and OpenMP has the following characteristics:

- The program has 6 MPI processes
- Each process uses 4 threads

The program has to run in a cluster that has 6 nodes; each node has 2 CPUs and each CPU has 8 cores.

- How many instances of the program can be executed simultaneously at the system?
- What values should be set in the corresponding SLURM script so that the program is executed using the adequate number of resources AND the maximum number of instances of the program can be executed simultaneously?
  (Specify the values of the required options among these ones: --ntasks, --nodes, --tasks-per-node, --cpus-per-task, --ntasks-per-core, --ntasks-per-node, --ntasks-per-socket, --threads-per-core, --cores-per-socket, --sockets-per-node)

We have 96 cores.
The program needs 24 threads

Then, 4 instances of the program can be executed simultaneously at the system

-- ntasks → 6	**# Number of MPI processes**
-- nodes→ 6	**# Number of nodes**
-- tasks-per-node → 1	# Number of MPI processes per node
-- cpus-per-task → 4	**# Number of threads per MPI process**

```
#SBATCH --nodes=6              # Number of nodes
#SBATCH --ntasks=12            # Number of MPI processes (6 nodes * 2 CPUs per node)
#SBATCH --cpus-per-task=4      # Number of threads per MPI process
#SBATCH --ntasks-per-node=2    # Number of MPI processes per node
#SBATCH --threads-per-core=2   # Number of threads per CPU core
```

--nodes=6 specifies that 6 nodes are requested.
--ntasks=12 specifies that there are 12 MPI processes (2 CPUs per node * 6 nodes).
--cpus-per-task=4 specifies that each MPI process will use 4 threads.
--ntasks-per-node=2 specifies that there are 2 MPI processes per node, each assigned to a separate CPU.
--threads-per-core=2 specifies that each CPU core will have 2 threads.

3.- (3 points)
The following program searches the biggest element in positions i..N-1 from vector values[] and the results are stored in a second vector (maxs[]). Parallelize this program by adding the appropriate OpenMP statements. Provide two different solutions: one based on the parallelization of the outer loop and the second one based on the parallelization of the inner loop. Add variables and changes in the code in order to achieve a more efficient solution (the score of the exercise will take into account this).

Provide explanations that describe how the program is parallelized and how threads are going to collaborate in each case, assuming the application will run with T threads (T<<N).

If you do not know the syntax or the specific name of a directive, add some explanatory comments to refer to it and its role within the program.

```
int values[N], sums[N]
int i , j, s;

for (i = 0; i < N ; i ++) {
     s = values[i];
     for (j = i+1; j < N ; j ++)
           if (s < values[j])
              s = value[j];
     maxs[i] = s;
}
```

```
int values[N], maxs[N]
int s;

#pragma omp parallel for schedule(dynamic, 1) private(s)
for (int i = 0; i < N ; i ++) {
   s = values[i];

   for (int j = i+1; j < N ; j ++)
      if (s < values[j])
          s = value[j];

   maxs[i] = s;
}
```

Given the following program that has been parallelized with OpenMP directives…

```
int  main ( ) {
int tmp =5;
 #pragma omp parallel for num_threads (4)
    for ( int j= 0 ; j <8; ++j )
         tmp = tmp+j ;
printf ( "Final value of tmp =%d \n" , tmp ) ;
}
```

What value will be printed in each of the following cases, when the corresponding directive(s) are added at the openmp clause?

a) private (tmp)
b) firstprivate (tmp) lastprivate (tmp)
c) reduction (+ : tmp)

private(tmp) → 5
firstprivate(tmp) → 5
lastprivate(tmp) → 18
reduction(+:tmp) → 33

Given the following program, write the data directive that should be used to transfer arrays between the CPU and the GPU (in both directions). Include also the array shaping information. Try to use clauses that minimize the total amount of data transferred (the score of the exercise will take into account this).

```
float array_sum[NX], u[NX][NY];


<---initialization of array_sum and u--->


# pragma acc parallel
    for (i = 1; i < NX + 1; i++)
        for (j = 1; j < NY + 1; j++)
            array_sum[i] += u[i][j];
```

```
#pragma acc data copyin(u[1:NX][1:NY]) copyout(array_sum[1:NX])
{
        #pragma acc parallel
        …
}
```

6.- (1,5 points)
Given the following loops, do they exhibit dependencies that prevent parallelization? If yes, are there simple transformations that can be applied to enable parallelization? Provide a brief explanation of your answers and the potential transformations.

```
//***** Loop 1 ***********

  for (int i = 1 ; i < N-1; i++)
     v[i+1]  = v[i] + w[i];

//***** Loop 2  ***********

  for (int i = 0 ; i < N-1; i++)
     v[i]  = v[i+1] + v[i];

//***** Loop 3 ***********

  for (int i = 1 ; i < N-1; i++)
  {
      v[i] = w[i+1] + E;
      y[i-1] =  v[i]  + C;
  }
```

Loop 1: Loop carried dependence. Recurrence pattern.

Loop 2: Anti dependence because one iteration requires data that would be modified later in the serial case, implying an order that is not there in the parallel case. It could be solved with an auxiliary array (W array is a copy of V array).

Loop 3: No dependency

7.- (1 point)

Which parallel pattern best fits each of the following problems? Justify it briefly.

a- Finding the maximum value of a list of N numbers.

b- Simulating heat transfer in one dimension surface in which temperature of each point at time t is computed considering its temperature and the temperature of its surrounding points at time t-1.

c- Adding a given constant to a set of N numbers.

a) **Reduction** → We can combine every element using an associative “combiner function”

b) **Stencil** → I want to compute an element and to do it, I depend on some of the neighbors.

c) **Map** → Performs a computation over every element of a collection that is independent.

Exam June 2020

1.- (4 points)
Consider the following program. For each element a[i] the program counts the number of elements that are less than a[i] and bigger than a[i]. The computed values are finally stored in two lists. Parallelize this program by adding the appropriate OpenMP statements. Provide two different solutions: one based on the parallelization of the outer loop and the second one based on the parallelization of the inner loop.

Provide explanations that describe how the program is parallelized and how threads are going to collaborate in each case, assuming the application will run with T threads.

```
int a [N], big [N], small[N]
int i , j , less, great ;

for ( i = 0; i < N ; i ++) {
   great = 0;
   less = 0;
   for ( j = 0; j < N ; j ++)
      if ( a [ j ] > a [ i ])
         less ++;
      if ( a [ j ] < a [ i ] )
         great ++;
   big [ i ] = great;
   small [ i ] = less;
}
```

```
int a [N], big [N], small[N]
int less, great ;

#pragma omp parallel for schedule(dynamic, 1) private(less, great)
for ( int i = 0; i < N ; i ++) {
   great = 0;
   less = 0;

   for ( int j = 0; j < N ; j ++)
      if ( a [ j ] > a [ i ])
         less ++;
      if ( a [ j ] < a [ i ] )
         great ++;

   big [ i ] = great;
   small [ i ] = less;
}
```

2.- (2 points)
Using the same sequential program from Exercise 1, parallelize it by adding the appropriate OpenACC statements. Provide a single solution that parallelizes both loops. Use the *kernels* construct and add all necessary data clauses. If needed, add additional directives or clauses to guarantee a correct execution.
Provide a explanation that describes how the program is parallelized and how data is moved between the host and the GPU.
If you do not know the syntax or the specific name of a directive, add some explanatory comments to refer to it and its role within the program.

```
int a [N], big [N], small[N]
int less, great ;

#pragma acc data copyin(a[:N]) copyout(big[:N], small[:N])
#pragma acc kernels {
for ( int i = 0; i < N ; i ++) {
    great = 0;
    less = 0;

    for ( int j = 0; j < N ; j ++)
        if ( a [ j ] > a [ i ])
            less ++;
        if ( a [ j ] < a [ i ] )
            great ++;

    big [ i ] = great;
    small [ i ] = less;
}
}
```

3.- (2 points)
Given the following four loops, which are vectorizable and which are not. If a loop cannot be vectorized, what's the reason? Provide a brief explanation.

```
//*****  Loop  1  ***********

  for (int i = 1 ; i < N; i++)
    v[i-1]  = v[i] + w[i-1];

//*****  Loop 2   ***********

  for (int i = 0 ; i < N-1; i++)
    v[i+1]  = v[i] + w[i+1];

//*****  Loop 3   ***********

  for (int i = 0 ; i < N-1; i++)
  {
    v[i+1] = w[i] + C;
    y[i] =  v[i]  + E;
  }

 //*****  Loop 4   ***********

  for (int i = 0 ; i < N-1; i++)
    v[i+1]  = v[i] +C;
```

Loop 1: No because there is an anti dependence. Because one iteration requires data that would be modified later in the serial case, implying an order that is not there in the parallel case.

Loop 2: No because there is a loop carried dependency. Because one iteration requires data that has been modified previously, implying an order that is not there in the parallel case.

Loop 3: No, because to compute Y[i] we need to previously have computed v[i+1], so this implies an order that is not there in the parallel case.

Loop 4: No because there is a loop carried dependency. Because one iteration requires data that has been modified previously, implying an order that is not there in the parallel case.

```
#define N 8

omp_set_num_threads(2);

#pragma omp parallel for ordered (1) schedule (static,2)

for (int i=2; i<N; i++) {
   printf ("Initial THR  %d  IT  %d\n", omp_get_thread_num(), i);
   #pragma omp ordered depend(sink: i-1)
    {
       printf ("Second THR %d IT  %d\n", omp_get_thread_num(), i);
     }
     #pragma omp ordered depend(source)
     printf ("Last THR %d IT %d \n", omp_get_thread_num(), i);
}
```

Indicate which outputs generated by the program would be possible and which would be impossible. Reason briefly when impossible.

Initial THR 0 IT 2	Initial THR 0 IT 2	Initial THR 0 IT 2	Initial THR 0 IT 2
Initial THR 1 IT 3	Second THR 0 IT 2	Second THR 0 IT 2	Second THR 0 IT 2
Second THR 0 IT 2	Initial THR 1 IT 3	Last THR 0 IT 2	Initial THR 0 IT 3
Last THR 0 IT 2	Last THR 0 IT 2	Initial THR 0 IT 4	Last THR 0 IT 2
Initial THR 0 IT 4	Initial THR 0 IT 4	Initial THR 1 IT 3	Initial THR 1 IT 4
Second THR 1 IT 3	Second THR 1 IT 3	Second THR 1 IT 3	Second THR 1 IT 4
Last THR 1 IT 3	Second THR 0 IT 4	Initial THR 1 IT 5	Second THR 0 IT 3
Initial THR 1 IT 5	Last THR 1 IT 3	Last THR 1 IT 3	Last THR 1 IT 4
Second THR 0 IT 4	Initial THR 1 IT 5	Second THR 0 IT 4	Last THR 0 IT 3
Second THR 1 IT 5	Last THR 0 IT 4	Last THR 0 IT 4	Initial THR 1 IT 5
Last THR 1 IT 5	Second THR 1 IT 5	Initial THR 0 IT 6	Initial THR 0 IT 6
Last THR 0 IT 4	Initial THR 0 IT 6	Second THR 1 IT 5	Second THR 0 IT 6
Initial THR 0 IT 6	Last THR 1 IT 5	Second THR 0 IT 6	Last THR 0 IT 6
Initial THR 1 IT 7	Initial THR 1 IT 7	Last THR 0 IT 6	Second THR 1 IT 5
Second THR 0 IT 6	Second THR 0 IT 6	Initial THR 1 IT 7	Last THR 1 IT 5
Second THR 1 IT 7	Last THR 0 IT 6	Last THR 1 IT 5	Initial THR 0 IT 7
Last THR 1 IT 7	Second THR 1 IT 7	Second THR 1 IT 7	Second THR 0 IT 7
Last THR 0 IT 6	Last THR 1 IT 7	Last THR 1 IT 7	Last THR 0 IT 7
(A)	(B)	(C)	(D)

Options A, B and C are not possible because they do not follow a static schedule with chunks of size 2.

Option D follows a static schedule with chunks of size 2 but does not follow the dependencies.

Exam July 2020

1.- (4 points)
Consider the following program. For each element values[i] the program adds the elements in positions i..N. The computed values are finally stored in one list. Parallelize this program by adding the appropriate OpenMP statements. Provide two different solutions: one based on the parallelization of the outer loop and the second one based on the parallelization of the inner loop. Add variables and changes in the code in order to achieve a more efficient solution (the score of the exercise will take into account this).

Provide explanations that describe how the program is parallelized and how threads are going to collaborate in each case, assuming the application will run with T threads.

If you do not know the syntax or the specific name of a directive, add some explanatory comments to refer to it and its role within the program.

```
int values [N], sums [N]
int i , j, addition;

for ( i = 0; i < N ; i ++) {
   sums[i] = values[i];
   for ( j = i; j < N ; j ++)
        sums[i] = values[j];
}
```

```
int values [N], sums [N]

#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < N ; i ++) {
   sums[i] = values[i];

   for (int  j = i; j < N ; j ++)
      sums[i] += values[j];
}
```

2.- (2 points)
Using the same sequential program from Exercise 1, parallelize it by adding the appropriate OpenACC statements. Provide a single solution that parallelizes both loops. Use the *kernels* construct and add all necessary data clauses. If needed, add additional directives or clauses to guarantee a correct execution.
Provide a explanation that describes how the program is parallelized and how data is moved between the host and the GPU.
If you do not know the syntax or the specific name of a directive, add some explanatory comments to refer to it and its role within the program.

```
int values [N], sums [N]
int addition;

#pragma acc data copyin(values[:N]) copyout(sums[:N])
#pragma acc kernels {
for (int i = 0; i < N ; i ++) {
    sums[i] = values[i];

    for (int  j = i; j < N ; j ++)
        sums[i] += values[j];
}
}
```

3.- (2 points)
Given the following four loops, which are vectorizable and which are not. If a loop cannot be vectorized, what's the reason? Provide a brief explanation.

```
//*****  Loop 1   ***********

  for (int i = 1 ; i < N-1; i++)
    v[i]  = v[i-1] + w[i+1];

//*****  Loop 2   ***********

  for (int i = 1 ; i < N-1; i++)
    v[i+1]  = v[i-1] +C;

//*****  Loop  3  ***********

  for (int i = 1 ; i < N; i++)
    v[i-1]  = v[i+1] + w[i-1];

//*****  Loop 4   ***********

  for (int i = 0 ; i < N-1; i++)
  {
    y[i] =  v[i]  + C;
    v[i+1] = w[i] + E;
  }
```

Loop 1: There is a loop carried dependency, so it can not be parallelized.

Loop 2: There is a loop carried dependency, so it can not be parallelized.

Loop 3: There is an anti dependence, so it can not be parallelized.

Loop 4: There is loop carried dependency, so it can not be parallelized.

```
#define N 8

omp_set_num_threads(2);

#pragma omp parallel for ordered (1) schedule (static,1)

for (int i=2; i<N; i++) {
   printf ("Initial THR  %d  IT  %d\n", omp_get_thread_num(), i);
   #pragma omp ordered depend(sink: i-1)
    {
       printf ("Second THR %d IT  %d\n", omp_get_thread_num(), i);
     }
     #pragma omp ordered depend(source)
     printf ("Last THR %d IT %d \n", omp_get_thread_num(), i);
}
```

Indicate which outputs generated by the program would be possible and which would be impossible. Reason briefly when impossible.

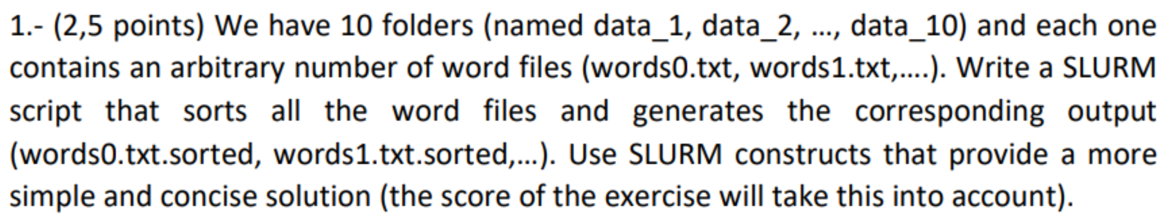
Initial THR 0 IT 2	Initial THR 0 IT 2	Initial THR 0 IT 2	Initial THR 0 IT 2
Initial THR 1 IT 3	Second THR 0 IT 2	Second THR 0 IT 2	Second THR 0 IT 2
Second THR 0 IT 2	Initial THR 1 IT 3	Last THR 0 IT 2	Initial THR 0 IT 3
Last THR 0 IT 2	Last THR 0 IT 2	Initial THR 0 IT 4	Last THR 0 IT 2
Initial THR 0 IT 4	Initial THR 0 IT 4	Initial THR 1 IT 3	Initial THR 1 IT 4
Second THR 1 IT 3	Second THR 1 IT 3	Second THR 1 IT 3	Second THR 1 IT 4
Last THR 1 IT 3	Second THR 0 IT 4	Initial THR 1 IT 5	Second THR 0 IT 3
Initial THR 1 IT 5	Last THR 1 IT 3	Last THR 1 IT 3	Last THR 1 IT 4
Second THR 0 IT 4	Initial THR 1 IT 5	Second THR 0 IT 4	Last THR 0 IT 3
Second THR 1 IT 5	Last THR 0 IT 4	Last THR 0 IT 4	Initial THR 1 IT 5
Last THR 1 IT 5	Second THR 1 IT 5	Initial THR 0 IT 6	Initial THR 0 IT 6
Last THR 0 IT 4	Initial THR 0 IT 6	Second THR 1 IT 5	Second THR 0 IT 6
Initial THR 0 IT 6	Last THR 1 IT 5	Second THR 0 IT 6	Last THR 0 IT 6
Initial THR 1 IT 7	Initial THR 1 IT 7	Last THR 0 IT 6	Second THR 1 IT 5
Second THR 0 IT 6	Second THR 0 IT 6	Initial THR 1 IT 7	Last THR 1 IT 5
Second THR 1 IT 7	Last THR 0 IT 6	Last THR 1 IT 5	Initial THR 0 IT 7
Last THR 1 IT 7	Second THR 1 IT 7	Second THR 1 IT 7	Second THR 0 IT 7
Last THR 0 IT 6	Last THR 1 IT 7	Last THR 1 IT 7	Last THR 0 IT 7
(A)	(B)	(C)	(D)

Option D is not possible because they do not follow a static schedule with chunks of size 1.

Option A and B are possible.

Option C is not possible because thread 1 is doing at the same time 2 iterations (3 and 5)

Exam July 2022


1.- (2,5 points) We have 10 folders (named data_1, data_2, …, data_10) and each one contains an arbitrary number of word files (words0.txt, words1.txt,….). Write a SLURM script that sorts all the word files and generates the corresponding output (words0.txt.sorted, words1.txt.sorted,…). Use SLURM constructs that provide a more simple and concise solution (the score of the exercise will take this into account).

We need to make a bash script that iterates over all files of each of the folders.

```
## TODO EN UN SLURM SCRIPT
#!/bin/bash
#SBATCH --job-name=basic-job
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --partition=nodo.q

path = “path to the parent directory”

for dir in $path/* do
        cd dir
        for file in directory do
                basename="${file%.*}"
                nfile="$basename.sorted"
                sort $file > $nfile
        done
        cd ..
done
```

2.- (1,5 points) What will be the output of the following code?

```
int  a, tid;
int main ()
{
  printf("1st Parallel Region:\n");
  #pragma omp parallel private(tid) num_threads(3)
  for (int i = 0; i < 3; i++) {
      tid = omp_get_thread_num();
      printf("Thread %d: doing one iteration \n",tid);
  }
  printf("************************************\n");
  printf("2nd Parallel Region:\n");
  #pragma omp parallel for private(tid) num_threads(3)
  for (int i = 0; i < 3; i++) {
     tid = omp_get_thread_num();
     printf("Thread %d: doing one iteration \n",tid);
  }
}
```

```
1st Parallel Region
Thread 0: doing one iteration
Thread 0: doing one iteration
Thread 0: doing one iteration
Thread 1: doing one iteration          # These 3 lines can be in any order…
Thread 1: doing one iteration
Thread 1: doing one iteration
Thread 2: doing one iteration
Thread 2: doing one iteration
Thread 2: doing one iteration
***************************
2nd Parallel Region
Thread 0: doing one iteration
Thread 1: doing one iteration          # These 3 lines can be in any order…
Thread 2: doing one iteration
```

3. – (1,5 points) Which combinations are correct for the pragma directive included in the following code? Explain briefly why the rest are not correct.

a) # pragma omp shared(tmp) reduction(+:sum)
b) # pragma omp shared(sum) reduction(+:tmp)
c) # pragma omp private(tmp) reduction(+:sum)
d) # pragma omp private(sum) reduction(+:tmp)

```
double work( double *a, double *b, int n )
{
  int i;
  double tmp, sum;
  sum = 0.0;
 #pragma omp parallel for private(?) reduction(?)
  for (i = 0; i < n; i++) {
      tmp = a[i] + b[i];
      sum += tmp;
   }
   return sum;
}
```

The correct pragma directive is C.

4.- (2,5 points) Consider the following program. For each element, values[i], the program counts the number of elements that are bigger and smaller than values[i]. The computed values are finally stored in two arrays B and S. Parallelize this program by adding the appropriate OpenMP statements.

```
 int values[N], B[N], S[N]
 int i, j, big, small;

for ( i = 0; i < N ; i ++) {
     big = 0;
     small = 0;
     for ( j = 0; j < N ; j ++) {
          if (values[j] > values[i])
             big++;
          if (values[j] < values[i])
             small++;
     }
     B[i] = big;
     S[i] = small;
 }
```

```
int values[N], B[N], S[N]
int big, small;

#pragma omp parallel for schedule(dynamic, 1)
for (int i = 0; i < N ; i ++) {
   big = 0;
   small = 0;

   #pragma omp parallel for schedule(dynamic, 1) reduction(+: big, +: small)
   for (int j = 0; j < N ; j ++) {
      if (values[j] > values[i])
         big++;
      if (values[j] < values[i])
         small++;
      }

 B[i] = big;
 S[i] = small;
}
```

5.- (2 points)
a) Name two mechanisms that can be used to specify the desired number of threads in an OpenMP application.

b) Which OpenACC pragma is used to:

- copy data from the host to the GPU
- allocate data elements that are only used at the GPU
- copy data from the GPU to the host

c) In OpenACC, which is the difference between the *kernels* directive and the *parallel* directive?

d) Explain briefly the parallel patterns of scan and map.

a) Mechanisms to specify the number of threads:
   i) Environment variable → OMP_NUM_THREADS=X ./name_file
   ii) Use a function → omp_set_num_threads(X);
   iii) Use a pragma directive → num_threads(X)

b) OpenACC pragmas:
   i) Copyin
   ii) Create
   iii) Copyout

c) When using kernels directive, we are telling the compiler that we want to parallelize this loop and the compiler will do it automatically, assuming all the responsibility (the final decision is taken by the compiler). But, if we use parallel directive, we will have more control. Meaning that we are specifically saying that we want to parallelize that part of the code, assuming all responsibilities.

d) **Map**: Performs a computation over every element of a collection that is independent.
   Scan: Computes all partial reductions of a collection.