1 Loop parallelism

Question 1–1

According to the Bernstein conditions, indicate the type of data dependency among the different iterations in the cases shown below. Justify if these data dependencies can be eliminated or not, removing it if possible.

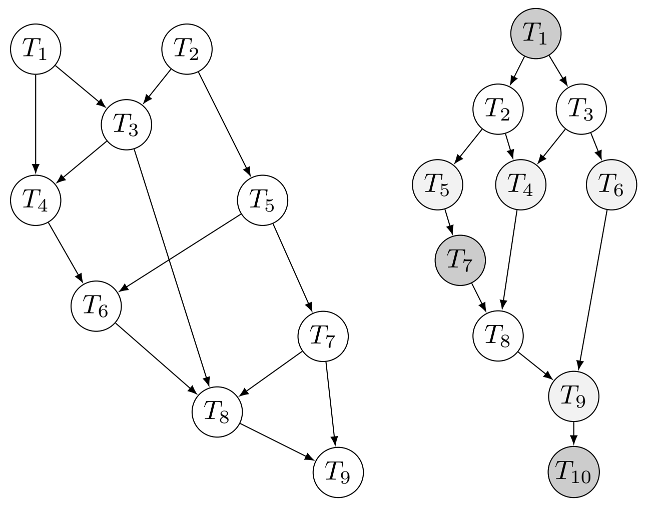
(a)
```
for (i=1;i<N-1;i++) {
  x[i+1] = x[i] + x[i-1];
}
```

(b)
```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
  x = a[i];
}
```

(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

Question 1–2

Given the following two task dependency graphs:



T_1, T_7, T_{10}	$\frac{1}{3}n^3$
T_2, T_3, T_8	n^3
T_4, T_5, T_6, T_9	$2n^3$

(a) For the left graph, indicate which sequence of graph nodes constitute the critical path. Compute the length of the critical path and the average concurrency degree. Note: no cost information is provided, one can assume that all tasks have the same cost.

(b) Repeat the same for the graph on the right. Note: in this case the cost of each task is given in flops (for a problem size n) according to the table shown.

Question 1–3

The following sequential code implements the product of an $N \times N$ matrix B by a vector c of dimension N.

```
void prodmv(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  for (i=0; i<N; i++) {
    sum = 0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(a) Create a parallel implementation of the above code with OpenMP.

(b) Compute the computational cost in flops of the sequential and parallel versions, assuming that the number of threads p is a divisor of N.

(c) Compute the speedup and efficiency of the parallel code.

Question 1–4

Given the following function:

```
double function(double A[M][N])
{
  int i,j;
  double sum;
  for (i=0; i<M-1; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = 2.0 * A[i+1][j];
    }
  }
  sum = 0.0;
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      sum = sum + A[i][j];
    }
  }
  return sum;
}
```

(a) Calculate the theoretical cost (in flops).

(b) Implement a parallel version using OpenMP. Justify any modifications that you make. Efficiency of the proposed solution will be taken into account.

(c) Calculate the speed-up that can be achieved with p processors assuming that M and N are exact multiples of p.

(d) Give an upper value for the speed-up (when p tends to infinity) in the case that only the first part is performed in parallel and the second part (the one related to the sum) is performed sequentially.

Question 1–5

Given the following function:

```
double fun_mat(double a[n][n], double b[n][n])
{
  int i,j,k;
  double aux,s=0.0;
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      aux=0.0;
      s += a[i][j];
      for (k=0; k<n; k++) {
        aux += a[i][k] * a[k][j];
      }
      b[i][j] = aux;
    }
  }
  return s;
}
```

(a) Describe how each of the three loops can be parallelised using OpenMP. Which will be the most efficient one? Justify your answer.

(b) Assuming that only the outer loop is parallelised, indicate the a priori sequential and parallel costs in flops. Calculate also the speed-up assuming that the number of threads (and processors) is n.

(c) Add the code lines required for showing on the screen the number of iterations that thread 0 has performed, in the case that the outer loop is parallelised.

Question 1–6

Implement a parallel program using OpenMP that satisfies the following requirements:

- Requests the user to enter a positive integer number n.
- Computes in parallel the sum of the first n natural numbers, using a dynamic distribution with chunk size equal to 2, and using 6 threads.
- At the end the program must print the identifier of the thread that summed the last number (n) as well as the computed sum.

Question 1–7

We want to efficiently parallelize the following function by means of OpenMP.

```
#define EPS 1e-9
#define N 128
int fun(double a[N][N], double b[], double x[], int n, int nMax)
{
  int i, j, k;
  double err=100, aux[N];

  for (i=0;i<n;i++)
     aux[i]=0.0;

  for (k=0;k<nMax && err>EPS;k++) {
     err=0.0;
     for (i=0;i<n;i++) {
        x[i]=b[i];
        for (j=0;j<i;j++)
           x[i]-=a[i][j]*aux[j];
        for (j=i+1;j<n;j++)
          x[i]-=a[i][j]*aux[j];
        x[i]/=a[i][i];
        err+=fabs(x[i]-aux[i]);
      }
      for (i=0;i<n;i++)
        aux[i]=x[i];
    }
    return k<nMax;
  }
```

(a) Parallelize it efficiently.

(b) Compute the computational cost of a single iteration of loop k. Compute the computational cost of the parallel version (assuming that the number of iterations divides the number of threads exactly) and the speed-up.

Question 1–8

Given the following function:

```
#define N 6000
#define STEPS 6

double function1(double A[N][N], double b[N], double x[N])
{
  int i, j, k, n=N, steps=STEPS;
  double max=-1.0e308, q, s, x2[N];
  for (k=0;k<steps;k++) {
    q=1;
    for (i=0;i<n;i++) {
      s = b[i];
      for (j=0;j<n;j++)
        s -= A[i][j]*x[j];
      x2[i] = s;
      q *= s;
    }
    for (i=0;i<n;i++)
      x[i] = x2[i];
    if (max<q)
      max = q;
  }
  return max;
}
```

(a) Parallelize the code using OpenMP. Explain why you do it that way. Those solutions that take into account efficiency will get a higher mark.

(b) Indicate the theoretical cost (in flops) of an iteration of the `k` loop in the sequential code.

(c) Considering a single iteration of the `k` loop (`STEPS`=1), indicate the speedup and efficiency that can be attained with p threads, assuming that there are as many cores/processors as threads and that `N` is an exact multiple of p.

Question 1–9

Given the following function:

```
void func(double A[M][P], double B[P][N], double C[M][N], double v[M]) {
   int i, j, k;
   double mf, val;
   for (i=0; i<M; i++) {
      mf = 0;
      for (j=0; j<N; j++) {
         val = 2.0*C[i][j];
         for (k=0; k<i; k++) {
            val += A[i][k]*B[k][j];
         }
         C[i][j] = val;
         if (val<mf) mf = val;
      }
      v[i] += mf;
   }
}
```

(a) Implement a parallel version based on the parallelisation of the `i` loop.

(b) Implement a parallel version based on the parallelisation of the `j` loop.

(c) Obtain the *a-priori* sequential execution time of a single iteration of the `i` loop, as well as the *a-priori* sequential execution time of the entire function. Let's suppose that the cost of comparing two floating point numbers is 1 flop.

(d) Analyse if there would be a good load balance if the clause `schedule(static)` is used in the parallelisation of the loop of the first part. Reason the answer.

Question 1–10

Given the next function:

```
double cuad_mat(double a[N][N], double b[N][N])
{
   int i,j,k;
   double aux, s=0.0;
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         aux = 0.0;
         for (k=i; k<N; k++)
            aux += a[i][k] * a[k][j];
         b[i][j] = aux;
         s += aux*aux;
      }
   }
   return s;
}
```

(a) Implement an efficient parallel version of the previous code using OpenMP. Out of the potential schedulings, which would be the most efficient ones? Reason your answer.

(b) Obtain the sequential cost in flops of the algorithm.

Question 1–11

Given the following function:

```
double f(double A[N][N], double B[N][N], double vs[N], double bmin) {
  int i, j;
  double x, y, aux, stot=0;
  for (i=0; i<N; i++) {
    aux = 0;
    for (j=0; j<N; j++) {
      x = A[i][j]*A[i][j]/2.0;
      A[i][j] = x;
      aux += x;
    }
    for (j=i; j<N; j++) {
      if (B[i][j]<bmin) y = bmin;
      else              y = B[i][j];
      B[i][j] = 1.0/y;
    }
    vs[i] = aux;
    stot += vs[i];
  }
  return stot;
}
```

(a) Paralellize (efficiently) the `i` loop by means of OpenMP.

(b) Paralellize (efficiently) both `j` loops by means of OpenMP.

(c) Compute the sequential cost of the original code.

(d) Assuming that we parallelize only the first `j` loop, compute the parallel cost of such version. Obtain the speedup and efficiency in case we have `N` processors available.

2 Parallel regions

Question 2–1

Assuming the following function, which searches for a value in a vector, implement a parallel version using OpenMP. As in the original function, the parallel function should end as soon as the sought element is found.

```
int search(int x[], int n, int value)
{
   int found=0, i=0;
   while (!found && i<n) {
      if (x[i]==value) found=1;
      i++;
   }
   return found;
}
```

Question 2–2

Given a vector v of n elements, the following function computes its 2-norm $\|v\|$, defined as:

$$\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

```
double norm(double v[], int n)
{
   int i;
   double r=0;
   for (i=0; i<n; i++)
      r += v[i]*v[i];
   return sqrt(r);
}
```

(a) Implement a parallel version of the function using OpenMP, and following this scheme:

- In a first stage, each thread computes the sum of the squares in a n/p block of vector v (given that p is the number of threads). Each thread will store the result of its part in the corresponding position of a vector `sums` with p elements. Assume that vector `sums` is already created, although not yet initialized.
- In a second stage, one of the threads will compute the norm of the vector from the individual results stored in the vector `sums`.

(b) Implement a parallel version of the function using OpenMP, using your own (different from the previous one) approach.

(c) Calculate the a priori cost of the original sequential algorithm. Obtain the cost of the parallel algorithm from item a, and the associated speed-up. Provide a justification for the values.

Question 2–3

Given the following function:

```
void f(int n, double a[], double b[])
{
   int i;
   for (i=0; i<n; i++) {
      b[i]=cos(a[i]);
   }
}
```

Parallelize it, and have each thread write a message indicating its thread number and how many iterations it has processed. We want to show just one message per thread.

Question 2–4

Given the following function:

```
void normalize(double A[N][N])
{
  int i,j;
  double sum=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum = sum + A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(sum);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = factor*A[i][j];
    }
  }
}
```

(a) Implement a parallel version with OpenMP using two separated parallel regions (blocks).

(b) Parallelize it using a single OpenMP parallel region for both pairs of loops. In this case, could we use the `nowait` clause? Justify your answer.

Question 2–5

Given the following function:

```
double ej(double x[M], double y[N], double A[M][N])
{
   int i,j;
   double aux,s=0.0;
   for (i=0; i<M; i++)
      x[i] = x[i]*x[i];
   for (i=0; i<N; i++)
      y[i] = 1.0+y[i];
   for (i=0; i<M; i++)
      for (j=0; j<N; j++) {
         aux = x[i]-y[j];
         A[i][j] = aux;
         s += aux;
     }
   return s;
}
```

(a) Implement an efficient parallel version using OpenMP, using just one parallel region.

(b) Compute the number of flops of the initial solution and of the parallel version.

(c) Obtain the speed-up and the efficiency.

Question 2–6

Parallelize the following fragment of code by means of OpenMP sections. The second argument in functions `fun1`, `fun2` and `fun3` is an input-output parameter, that is, these functions use and modify the value of `a`.

```
int n=...;
double a,b[3];

a = -1.8;
fun1(n,&a);
b[0] = a;
a = 3.2;
fun2(n,&a);
b[1] = a;
a = 0.25;
fun3(n,&a);
b[2] = a;
```

Question 2–7

Given the following function:

```
void func(double a[],double b[],double c[],double d[])
{
  f1(a,b);
  f2(b,b);
```

```
     f3(c,d);
     f4(d,d);
     f5(a,a,b,c,d);
   }
```

The first argument in every used function is an output argument and the rest are input arguments. For instance, `f1(a,b)` is a function that modifies vector `a` from vector `b`.

(a) Draw the task dependency graph and indicate at least 2 different types of dependencies appearing in this problem.

(b) Parallize the function by means of OpenMP directives.

(c) Assuming that all functions have the same cost and that we have an arbitrary number of processors available, what will be the maximum possible speedup? Could this speedup be improved by means of data replication?

Question 2–8

In the following function, `T1`, `T2`, `T3` modify `x`, `y`, `z`, respectively.

```
double f(double x[], double y[], double z[], int n)
{
   int i, j;
   double s1, s2, a, res;

   T1(x,n);    /* Task T1 */
   T2(y,n);    /* Task T2 */
   T3(z,n);    /* Task T3 */

   /* Task T4 */
   for (i=0; i<n; i++) {
      s1=0;
      for (j=0; j<n; j++) s1+=x[i]*y[i];
      for (j=0; j<n; j++) x[i]*=s1;
   }

   /* Task T5 */
   for (i=0; i<n; i++) {
      s2=0;
      for (j=0; j<n; j++) s2+=y[i]*z[i];
      for (j=0; j<n; j++) z[i]*=s2;
   }

   /* Task T6 */
   a=s1/s2;
   res=0;
   for (i=0; i<n; i++) res+=a*z[i];
   return res;
}
```

(a) Draw the task dependency graph.

(b) Make a task-level parallelization by means of OpenMP (not a loop-level parallelization), based on the dependency graph.

(c) Indicate the a priori cost of the sequential algorithm, the parallel algorithm, and the resulting speedup. Suppose the cost of tasks 1, 2 and 3 is $2n^2$ flops each.

Question 2–9

Given the following fragment of a code:

```
minx = minimum(x,n);       /* T1 */
maxx = maximum(x,n);       /* T2 */
compute_z(z,minx,maxx,n);  /* T3 */
compute_y(y,x,n);          /* T4 */
compute_x(x,y,n);          /* T5 */
compute_v(v,z,x);          /* T6 */
```

(a) Draw a dependency graph of the tasks, taking into account that functions `minimum` and `maximum` do not change their arguments, and the rest of the functions only change the first argument.

(b) Implement a parallel version of the code using OpenMP.

(c) If the cost of the tasks is n flops (except for task 4 which takes $2n$ flops), calculate the length of the critical path and the average concurrency degree. Compute the speed-up and efficiency of the implementation written in the previous part, if 5 processors were used.

Question 2–10

We want to parallelize the following program by means of OpenMP, where `generate` is a function previously defined elsewhere.

```
double fun1(double a[],int n,
            int v0)
{
  int i;
  a[0] = v0;
  for (i=1;i<n;i++)
    a[i] = generate(a[i-1],i);
}
```

```
double compare(double x[],double y[],int n)
{
  int i;
  double s=0;
  for (i=0;i<n;i++)
    s += fabs(x[i]-y[i]);
  return s;
}
```

```
/* fragment of the main program */
int i, n=10;
double a[10], b[10], c[10], x=5, y=7, z=11, w;
fun1(a,n,x);           /* T1 */
fun1(b,n,y);           /* T2 */
fun1(c,n,z);           /* T3 */
x = compare(a,b,n);    /* T4 */
y = compare(a,c,n);    /* T5 */
z = compare(c,b,n);    /* T6 */
w = x+y+z;             /* T7 */
printf("w:%f\n", w);
```

(a) Parallelize the code efficiently at the level of the loops.

(b) Draw the task dependency graph, according to the numbering of tasks indicated in the code.

(c) Parallelize the code efficiently in terms of tasks, from the previous dependency graph.

(d) Obtain the sequential time (assume that a call to functions `generate` and `fabs` costs 1 flop) and the parallel time for each of the two versions assuming that there are 3 processors. Compute the speed-up in each case.

Question 2–11

Parallelize by means of OpenMP the following fragment of code, where `f` and `g` are two functions that take 3 arguments of type `double` and return a `double`, and `fabs` is the standard function that returns the absolute value of a `double`.

```
double x,y,z,w=0.0;
double x0=1.0,y0=3.0,z0=2.0;     /* starting point */
double dx=0.01,dy=0.01,dz=0.01;  /* increments */

x=x0;y=y0;z=z0;    /* search in x */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) x += dx;
w += (x-x0);

x=x0;y=y0;z=z0;    /* search in y */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) y += dy;
w += (y-y0);

x=x0;y=y0;z=z0;    /* search in z */
while (fabs(f(x,y,z))<fabs(g(x0,y0,z0))) z += dz;
w += (z-z0);

printf("w = %g\n",w);
```

Question 2–12

Considering the definition of the following functions :

```
/* Matrix product C = A*B */
void matmult(double A[N][N],
       double B[N][N],double C[N][N])
{
  int i,j,k;
  double sum;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum = 0.0;
      for (k=0; k<N; k++) {
        sum = sum + A[i][k]*B[k][j];
      }
      C[i][j] = sum;
    }
  }
}
```

```
/* Generate a symmetric matrix A+A' */
void symmetrize(double A[N][N])
{
  int i,j;
  double sum;
  for (i=0; i<N; i++) {
    for (j=0; j<=i; j++) {
      sum = A[i][j]+A[j][i];
      A[i][j] = sum;
      A[j][i] = sum;
    }
  }
}
```

we want to parallelize the following code:

```
matmult(X,Y,C1);    /* T1 */
matmult(Y,Z,C2);    /* T2 */
matmult(Z,X,C3);    /* T3 */
symmetrize(C1);     /* T4 */
symmetrize(C2);     /* T5 */
matmult(C1,C2,D1);  /* T6 */
matmult(D1,C3,D);   /* T7 */
```

(a) Implement a parallel version based on loops.

(b) Draw the task dependency graph, considering that in this case that each call to the `matmult` and `symmetrize` functions is an independent task. Indicate the maximum degree of concurrency, the length of the critical path and the average degree of concurrency. Note: to compute such values, you should obtain the cost in flops for both functions.

(c) Implement a parallelisation based on sections, according to the previous task dependency graph.

Question 2–13

Given the following function:

```
void updatemat(double A[N][N])
{
  int i,j;
  double s[N];
  for (i=0; i<N; i++) {      /* sum by rows */
    s[i] = 0.0;
    for (j=0; j<N; j++)
      s[i] += A[i][j];
  }
  for (i=1; i<N; i++)        /* prefix sum  */
    s[i] += s[i-1];
  for (j=0; j<N; j++) {      /* column scaling */
    for (i=0; i<N; i++)
      A[i][j] *= s[j];
  }
}
```

(a) Compute the theoretical cost (in flops) of the above function.

(b) Parallelise it with OpenMP using a single parallel region.

(c) Compute the speed-up that can be obtained using p processors and assuming that N is an exact multiple of p.

Question 2–14

Given the following function:

```
double calcula()
{
  double A[N][N],B[N][N],a,b,x,y,z;

  rellena(A,B);              /* T1 */
  a = calculos(A);           /* T2 */
  b = calculos(B);           /* T3 */
  x = suma_menores(B,a);     /* T4 */
  y = suma_en_rango(B,a,b);  /* T5 */
  z = x + y;                 /* T6 */
  return z;
}
```

Function `rellena` receives as input two matrices and fills them up with values generated internally. All the arguments in the rest of the functions are input parameters and are not modified. Functions `rellena` and `suma_en_rango` have a cost of $2n^2$ flops each ($n = \texttt{N}$), whereas the cost for the rest of the functions is of n^2 flops.

(a) Draw the dependency graph and compute its maximum degree of concurrency, the critical path and its length and the average concurrency degree.

(b) Parallelise the function using OpenMP.

(c) Compute the sequential execution time, the parallel execution time, the speed-up and the efficiency of the previous code, assuming that we have 3 threads.

Question 2–15

We want to parallelise the next code for processing images, which receives as input 4 similar images (e.g. video frames `f1`, `f2`, `f3`, `f4`) and returns two resulting images (`r1`, `r2`). The pixels in the images are represented as floating point values. Type `image` is a type defined that consists on a matrix of `N`×`M` `double`s.

```
typedef double image[N][M];

void procesa(image f1,image f2,image f3,image f4,image r1,image r2)
{
  image d1,d2,d3;
  difer(f2,f1,d1);          /* Task 1 */
  difer(f3,f2,d2);          /* Task 2 */
  difer(f4,f3,d3);          /* Task 3 */
  sum(d1,d2,d3,r1);         /* Task 4 */
  difer(f4,f1,r2);          /* Task 5 */
}
```

```
void difer(image a,image b,image d)
{
  int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      d[i][j] = fabs(a[i][j]-b[i][j]);
}
```

```
void sum(image a,image b,image c,image s)
{
  int i,j;
  for (i=0;i<N;i++)
    for (j=0;j<M;j++)
      s[i][j] = a[i][j]+b[i][j]+c[i][j];
}
```

(a) Draw the dependency graph of tasks, and compute the maximum and average concurrency degree, considering the actual cost in flops (assuming that `fabs` does not cost any flop).

(b) Parallelise the function `procesa` using OpenMP, without changing neither `difer` nor `sum`.

Question 2–16

In the next function (`matrix_computations`), no function call (`A`,`B`,`C`,`D`) changes any of its parameters:

```
double matrix_computations(double mat[n][n])
{
  double x,y,z,aux,total;
  x = A(mat);         /* task A, cost: 3 n^2         */
  aux = B(mat);       /* task B, cost: n^2           */
  y = C(mat,aux);     /* task C, cost: n^2           */
  z = D(mat);         /* task D, cost: 2 n^2         */
  total = x + y + z;  /* task E  (compute the cost)  */
  return total;
}
```

(a) Draw its dependency graph and compute the maximum concurrency degree, the length of the critical path (indicate a critical path) and the average concurrency degree.

(b) Implement a parallel version using OpenMP.

(c) Obtain the sequential time in flops. Assuming that the parallel version is executed with 2 threads, obtain the parallel time, the speed-up and the efficiency, in the best case.

(d) Modify the parallel cost to print on the screen (only once) the number of threads used and the execution time in seconds.

Question 2–17

Given the next function:

```
double function(double A[M][N], double maxim, double pf[])
{
   int i,j,j2;
   double a,x,y;
   x = 0;
   for (i=0; i<M; i++) {
      y = 1;
      for (j=0; j<N; j++) {
         a = A[i][j];
         if (a>maxim) a = 0;
         x += a;
      }
      for (j2=1; j2<i; j2++) {
         y *= A[i][j2-1]-A[i][j2];
      }
      pf[i] = y;
   }
   return x;
}
```

(a) Implement a parallel version based on the parallelisation of loop `i` using OpenMP.

(b) Implement another parallel version by parallelising the iterations of loops `j` and `j2` (efficiently and for any number of threads).

(c) Compute the sequential cost:

(d) For each one of the parallelised loops, analyse if you may expect differences in performance due to different scheduling policies. If so, write down the best scheduling for such loop.

Question 2–18

We want to implement a parallel version of the next function, where `read_data` updates its three arguments and `f5` reads and writes its two first arguments. The rest of the functions do not modify any of their arguments.

```
void function() {
  double x,y,z,a,b,c,d,e;
  int n;
  n = read_data(&x,&y,&z);      /* Task 1 (n flops)    */
  a = f2(x,n);                  /* Task 2 (2n flops)   */
  b = f3(y,n);                  /* Task 3 (2n flops)   */
  c = f4(z,a,n);                /* Task 4 (n^2 flops)  */
  d = f5(&x,&y,n);              /* Task 5 (3n^2 flops) */
  e = f6(z,b,n);                /* Task 6 (n^2 flops)  */
  write_results(c,d,e);         /* Task 7 (n flops)    */
}
```

(a) Draw the dependency graph for the different tasks involved in the function.

(b) Parallelise the function efficiently using OpenMP.

(c) Compute speed-up and efficiency if 3 processing units are used.

(d) Paying attention to the costs of each task (see the comments in the code of the function), obtain the critical path and the average concurrency degree.

Question 2–19

Given the following function, where all the function calls modify only the first vector received as an argument:

```
double f(double x[], double y[], double z[], double v[], double w[]) {
   double r1, res;
   A(x,v);              /* Task A. Cost of 2*n^2 flops */
   B(y,v,w);            /* Task B. Cost of   n   flops */
   C(w,v);              /* Task C. Cost of   n^2 flops */
   r1=D(z,v);           /* Task D. Cost of 2*n^2 flops */
   E(x,v,w);            /* Task E. Cost of   n^2 flops */
   res=F(z,r1);         /* Task F. Cost of 3*n   flops */
   return res;
}
```

(a) Draw the dependency graph. Identify a critical path and obtain its length. Calculate the average concurrency degree.

(b) Implement an efficient parallel version of the function.

(c) Let's suppose the code in the previous part is executed using only 2 threads. Compute the parallel execution cost, the speed-up and the efficiency in the best case. Reason the answer.

Question 2–20

In the next function none of the functions called modify their arguments.

```
int exercise(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
  a = task1(v,x);   /* task 1, cost n flops */
  b = task2(v,a);   /* task 2, cost n flops */
  c = task3(v,x);   /* task 3, cost 4n flops */
  x = x + a + b + c;   /* task 4 */
  for (i=0; i<n; i++) {   /* task 5 */
    j = f(v[i],x);   /* each function call costs 6 flops */
    if (j>0 && j<4) k++;
  }
  return k;
}
```

(a) Compute the sequential execution time.

(b) Draw a dependeny graph at the task level (considering task 5 as a whole), and obtain the maximum concurrency degree, the length of the critical path and the average concurrency degree.

(c) Implement an efficient parallel OpenMP version using a single parallel region. Parallelize both the tasks that could run concurrently and also the loop of task 5.

(d) Considering that the program is run using 6 threads (and assuming that n is an exact multiple of 6), obtain the parallel execution time, the speed-up and the efficiency.

Question 2–21

```
void matmult(double A[N][N],
       double B[N][N],double C[N][N]) {
  int i,j,k;
  double sum;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum = 0.0;
      for (k=0; k<N; k++) {
        sum += A[i][k]*B[k][j];
      }
      C[i][j] = sum;
    }
  }
}
```

```
void normalize(double A[N][N]) {
  int i,j;
  double sum=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum += A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(sum);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] *= factor;
    }
  }
}
```

Given the definition of the functions above, we want to parallelize the following code:

```
matmult(A,B,R1);    /* T1 */
matmult(C,D,R2);    /* T2 */
normalize(R1);      /* T3 */
normalize(R2);      /* T4 */
matmult(A,R2,M1);   /* T5 */
matmult(B,R2,M2);   /* T6 */
matmult(C,R1,M3);   /* T7 */
matmult(D,R1,M4);   /* T8 */
```

(a) Draw the task dependency graph. Indicate which is the length of the critical path and the average degree of concurrency. Note: to determine these values, it is necessary to obtain the cost in flops of both functions. Assume that `sqrt` costs 5 flops.

(b) Do the parallelization based on sections, using the previous task dependency graph.

Question 2–22

Given the following function:

```
double myadd(double A[N][M])
{
   double sum=0, maximum;
   int i,j;

   for (i=0; i<N; i++) {
      maximum=0;
      for (j=0; j<M; j++) {
         if (A[i][j]>maximum) maximum = A[i][j];
      }
      for (j=0; j<M; j++) {
         if (A[i][j]>0.0) {
            A[i][j] = A[i][j]/maximum;
            sum = sum + A[i][j];
         }
      }
   }
   return sum;
}
```

(a) Parallelize the function efficiently by means of OpenMP.

(b) Indicate its theoretical parallel cost (in flops), assuming that `N` is a multiple of the number of threads. To evaluate the cost consider the worst case, that is, when all comparisons are true. Also, suppose that the cost of comparing two real numbers is 1 *flop*.

(c) Modify the code so that each thread shows a single message with its thread number and the number of elements that it has summed.

Question 2–23

Given the following code:

```
double a,b,c,e,d,f;
T1(&a,&b); // Cost: 10 flops
c=T2(a);   // Cost: 15 flops
c=T3(c);   // Cost:  8 flops
d=T4(b);   // Cost: 20 flops
e=T5(c);   // Cost: 30 flops
f=T6(c);   // Cost: 35 flops
b=T7(c);   // Cost: 30 flops
```

(a) Obtain the task dependency graph and explain which type of dependencies occur between T_2 and T_3 and between T_4 and T_7, in case there is one.

(b) Compute the length of the critical path, and indicate the tasks that it contains.

(c) Implement a parallel version as efficient as possible of the previous code by means of sections, employing just one parallel region.

(d) Compute the speedup and efficiency in the case of using 4 threads to run the parallel code.

3 Synchronization

Question 3–1

Given the following code that allows sorting a vector `v` of `n` real numbers in ascending order:

```
int sorted = 0;
double a;
while( !sorted ) {
  sorted = 1;
  for( i=0; i<n-1; i+=2 ) {
    if( v[i]>v[i+1] ) {
      a = v[i];
      v[i] = v[i+1];
      v[i+1] = a;
      sorted = 0;
    }
  }
  for( i=1; i<n-1; i+=2 ) {
    if( v[i]>v[i+1] ) {
      a = v[i];
      v[i] = v[i+1];
      v[i+1] = a;
      sorted = 0;
    }
  }
}
```

(a) Introduce OpenMP directives that allow parallel execution of this code.

(b) Modify the code in order to count the number of exchanges, that is, the number of times that any of the two `if` clauses is entered.

Question 3–2

Given the function:

```
void f(int n, double v[], double x[], int ind[])
{
   int i;
   for (i=0; i<n; i++) {
      x[ind[i]] = max(x[ind[i]],f2(v[i]));
   }
}
```

Parallelize the function, taking into account that `f2` has very high cost. The proposed solution must be efficient.

Note. We assume that `f2` does not have lateral effects and its result only depends on its input argument. The return type of function `f2` is `double`. The function `max` returns the maximum of two numbers.

Question 3–3

Given the following function, that looks for a value in a vector

```
int search(int x[], int n, int value)
{
   int i, pos=-1;

   for (i=0; i<n; i++)
      if (x[i]==value)
         pos=i;

   return pos;
}
```

Parallelize it by means of OpenMP. In case of several occurrences of the value in the vector, the parallel algorithm must return the same results as the sequential one.

Question 3–4

The infinite-norm of a matrix $A \in \mathbb{R}^{n\times n}$ is defined as the maximum of the sum of the absolute values of the elements in each row:

$$\|A\|_\infty = \max_{i=0,\ldots,n-1} \left\{ \sum_{j=0}^{n-1} |a_{i,j}| \right\}$$

The following sequential code implements such operation for a square matrix.

```
#include <math.h>
#define DIMN 100

double infNorm(double A[DIMN][DIMN], int n)
{
  int i,j;
  double s,norm=0;

  for (i=0; i<n; i++) {
    s = 0;
    for (j=0; j<n; j++)
      s += fabs(A[i][j]);
    if (s>norm)
      norm = s;
  }
  return norm;
}
```

(a) Implement a parallel version of this algorithm using OpenMP. Justify each change introduced.

(b) Calculate the computational cost (in flops) for the original sequential version and for the parallel version implemented.
N.B.: Assume that the matrix dimension n is an exact multiple of the number of threads p. Assume that the computational cost for function `fabs` is 1 flop.

(c) Calculate the speed-up and efficiency of the parallel code when run with p processors.

Question 3–5

Given the following function, that computes the product of the elements of vector `v`:

```
double prod(double v[], int n)
{
  double p=1;
  int i;
  for (i=0;i<n;i++)
    p *= v[i];
  return p;
}
```

Implement two parallel functions:

(a) Using reduction.

(b) Without using reduction.

Implement two parallel functions that compute the factorial of a number:

(a) Using reduction.

(b) Without using reduction.

Question 3–6

The following code has to be efficiently parallelised using OpenMP.

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0;
  double aux;
  for (i=0; i<n; i++) {
    aux = x[i] - y[i];
    if (aux > 0) v = 1;
    else if (aux < 0) v = -1;
    else v = 0;
    z[i] = v;
    if (v == 0) equal++;
  }
  return equal;
}
```

(a) Implement a parallel version using only `parallel for` constructions.

(b) Implement a parallel version without using any of the following primitives: `for`, `section`, `reduction`.

Question 3–7

Given the following code fragment, where the vector of indices `ind` contains integer values between 0 and $m-1$ (being m the dimension of `x`), possibly with repetitions:

```
for (i=0; i<n; i++) {
   s = 0;
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   x[ind[i]] += s;
}
```

(a) Write a parallel implementation with OpenMP, in which the iterations of the outer loop are shared.

(b) Write a parallel implementation with OpenMP, in which the iterations of the inner loop are shared.

(c) For the implementation of item (a), indicate if we can expect performance differences depending on the schedule used. In this case, which schedule schemes would be better and why?

Question 3–8

The following function normalizes the elements of a vector of positive real numbers in a way that the end values remain between 0 and 1, using the maximum and the minimum.

```
void normalize(double *a, int n)
{
  double mx, mn, factor;
  int i;

  mx = a[0];
  for (i=1;i<n;i++) {
    if (mx<a[i]) mx=a[i];
  }
  mn = a[0];
  for (i=1;i<n;i++) {
    if (mn>a[i]) mn=a[i];
  }
  factor = mx-mn;
  for (i=0;i<n;i++) {
    a[i]=(a[i]-mn)/factor;
  }
}
```

(a) Parallelize the program with OpenMP in the most efficient way, by means of a single parallel region. We assume that the value of `n` is very large and we want that the parallelization works efficiently for an arbitrary number of threads.

(b) Include the necessary code so that the number of used threads is printed once.

Question 3–9

Given the function:

```
int function(int n, double v[])
{
  int i, max_pos=-1;
  double sum, norm, aux, max=-1;

  sum = 0;
  for (i=0;i<n;i++)
    sum = sum + v[i]*v[i];
  norm = sqrt(sum);

  for (i=0;i<n;i++)
    v[i] = v[i] / norm;

  for (i=0;i<n;i++) {
    aux = v[i];
    if (aux < 0)  aux = -aux;
    if (aux > max) {
      max_pos = i;  max = aux;
    }
  }
  return max_pos;
}
```

(a) Implement a parallel version using OpenMP, using a single parallel region.

(b) Would it be reasonable to use the `nowait` clause to any of the loops? Why? Justify each loop separately.

(c) What will you add to guarantee that all the iterations in all the loops are distributed in blocks of two iterations among the threads?

Question 3–10

The following function processes a series of bank transfers. Each transfer has an origin account, a destination account, and an amount of money that is moved from the origin account to the destination account. The function updates the amount of money in each account (`balance` array) and also returns the maximum amount that is transferred in a single operation.

```
double transfers(double balance[], int origins[], int destinations[],
       double quantities[], int n)
{
   int i, i1, i2;
   double money, maxtransf=0;

   for (i=0; i<n; i++) {
      /* Process transfer i: The transferred quantity is quantities[i],
       * that is moved from account origins[i] to account destinations[i].
       * Balances of both accounts are updated and the maximum quantity */
      i1 = origins[i];
      i2 = destinations[i];
      money = quantities[i];
      balance[i1] -= money;
      balance[i2] += money;
      if (money>maxtransf) maxtransf = money;
   }
   return maxtransf;
}
```

(a) Parallelize the function in an efficient way by means of OpenMP.

(b) Modify the previous solution so that the index of the transfer with more money is printed.

Question 3–11

Given the following function:

```
double function(double A[N][N],double B[N][N])
{
  int i,j;
  double aux, maxi;
  for (i=1; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = 2.0+A[i-1][j];
    }
  }
  for (i=0; i<N-1; i++) {
    for (j=0; j<N-1; j++) {
      B[i][j] = A[i+1][j]*A[i][j+1];
    }
  }
  maxi = 0.0;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      aux = B[i][j]*B[i][j];
      if (aux>maxi) maxi = aux;
    }
  }
  return maxi;
}
```

(a) Parallelize the previous code with OpenMP. Explain the decisions that you take. A higher consideration will be given to those solutions that are more efficient.

(b) Compute the sequential cost, the parallel cost, the speedup and the efficiency that could be obtained with p processors assuming that N is divisible by p.

Question 3–12

Next function computes all the row and column positions where the maximum value in a matrix appears.

```
int funcion(double A[N][N],double positions[][2])
{
  int i,j,k=0;
  double maximum;
  /* Compute maximum */
  maximum = A[0][0];
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j]>maximum) maximum = A[i][j];
    }
  }
  /* Once calculated, we seek for their occurrences */
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
      if (A[i][j] == maximum) {
        positions[k][0] = i;
        positions[k][1] = j;
        k = k+1;
      }
    }
  }
  return k;
}
```

(a) Parallelise this function efficiently using OpenMP, using a single parallel region.

(b) Modify the previous parallel code so that each thread prints on the screen its identifier and the amount of maximum values it found.

Question 3–13

Suppose there is a matrix M that stores data relative to the performance of the NJ players of a basketball team in different games. Each of the NA rows of the matrix corresponds to the performance of a player in a game, storing, in its 4 columns, the player's dorsal number (consecutive numbering from 0 to NJ-1), the number of points scored by the player in that game, the number of rebounds and the number of blocks. The individual valuation of a player in each game is calculated as follows:

$$\text{valuation} = \text{points} + 1.5 * \text{rebounds} + 2 * \text{blocks}$$

Parallelize with OpenMP with a single parallel region the following function in charge of obtaining and printing on the screen the player that has scored most points in a game, as well as of computing the average valuation of each player of the team.

```
void valuation(int M[][4], double average_valuation[NJ]) {
  int i,player,points,rebounds,blocks,max_points=0,max_scorer;
  double sum_valuation[NJ];
  int num_games[NJ];
  ...
  for (i=0;i<NA;i++) {
    player   = M[i][0];
    points   = M[i][1];
    rebounds = M[i][2];
    blocks   = M[i][3];
    sum_valuation[player] += points+1.5*rebounds+2*blocks;
    num_games[player]++;
    if (points>max_points) {
      max_points = points;
      max_scorer = player;
    }
  }
  printf("Maximum scorer: player %d (%d points)\n",max_scorer,max_points);
  for (i=0;i<NJ;i++) {
    if (num_games[i]==0)
      average_valuation[i] = 0;
    else
      average_valuation[i] = sum_valuation[i]/num_games[i];
  }
  ...
}
```

Question 3–14

In a photography contest, each member of the jury evaluates the photographs she or he considers relevant. We have implemented a function that receives all the scores given by all the members of the jury and a vector `totals` where the total score for each photograph will be computed. The vector `totals` is initially filled with zeros.

The function computes the total score for each photograph, showing on the screen the two highest scores given by any member of the jury. It also computes and displays on the screen the average score of all the photographs, as well as the number of photos that will pass to the next phase of the contest, which are the photos which got a score higher than or equal to 20 points.

The element `k` of vector `scores[k]` has the score of photo number `index[k]`. Obviously, a photo can obtain several scores from several members of the jury.

Implement an efficient parallel version using a single parallel OpenMP region.

```
/* nf = number of photos, nv = number of scores */
void contest(int nf, int totals[], int nv, int index[], int scores[])
{
  int k,i,p,t, pass=0, max1=-1,max2=-1, total=0;
  for (k = 0; k < nv; k++) {
    i = index[k]; p = scores[k];
    totals[i] += p;
    if (p > max2)
      if (p > max1) { max2 = max1; max1 = p; } else max2 = p;
  }
  printf("The two highest scores have been %d and %d.\n",max1,max2);
  for (k = 0; k < nf; k++) {
    t = totals[k];
    if (t >= 20) pass++;
    total += t;
  }
  printf("Average Score: %g. %d photos pass to the next round.\n",
   (float)total/nf, pass);
}
```

Question 3–15

The next function processes the billing, at the end of the month, of all the songs downloaded by a group of users in a music virtual shop. The shop registers the user identifier and the song identifier of each one of the n downloads performed. Those identifiers are stored in the vectors `users` and `songs` respectively. Each song has a different price, stored in the vector `prices`. The function also displays on the screen the identifier of the most downloaded song. The vectors `ndownloads` and `billing` are set to zero before the function is called.

```
void monthbilling(int n, int users[], int songs[], float prices[],
                   float billing[], int ndownloads[])
{
  int i,u,c,best_song=0;
  float p;
  for (i=0;i<n;i++) {
     u = users[i];
     c = songs[i];
     p = prices[c];
     billing[u] += p;
     ndownloads[c]++;
  }
  for (i=0;i<NC;i++) {
     if (ndownloads[i]>ndownloads[best_song])
        best_song = i;
  }
  printf("Song %d has been downloaded most\n",best_song);
}
```

(a) Implement an efficient parallel version in OpenMP of the above function using a single parallel region.

(b) Would it be reasonable to use the `nowait` clause in the first loop?

(c) Update the previous parallel code so that each thread displays on the screen the identifier and the number of iterations from the first loop that the thread has processed.

Question 3–16

We want to obtain the distribution of grades obtained by the CPA students, classifying the grades in five categories: *suspenso*, *aprobado*, *notable*, *sobresaliente* and *matrícula de honor*.

```
void histogram(int histo[], float marks[], int n) {
  int i, mark;
  float rmark;
  for (i=0;i<5;i++) histo[i] = 0;
  for (i=0;i<n;i++) {
    rmark = round(marks[i]*10)/10.0;
    if (rmark<5) mark = 0;        /* suspenso */
    else
      if (rmark<7) mark = 1;      /* aprobado */
      else
        if (rmark<9) mark = 2;    /* notable */
        else
          if (rmark<10) mark = 3; /* sobresaliente */
          else
            mark = 4;             /* matricula de honor */
    histo[mark]++;
  }
}
```

(a) Parallelize function `histogram` appropriately with OpenMP.

(b) Modify function `histogram` so that it prints the number of the student with the best mark and its mark, and the value of the worst mark (both of them without rounding).

Question 3–17

The following function manages a certain number of trips, that have taken place during a concrete period of time, by means of the public bicycle service of a city. For each of the trips, the identifiers of the source and destination stations are stored, together with the elapsed time (expressed in minutes) in each of them. The vector `num_bikes` stores the number of bicycles available in each station. Furthermore, the function computes between which stations the longest and shortest trips took place, together with the average elapsed time of all trips.

```
struct trip {
   int source_station;
   int dest_station;
   float time_minutes;
};
void update_bikes(struct trip trips[],int num_trips,int num_bikes[]) {
   int i,source,dest,srcmax,srcmin,destmax,destmin;
   float time,tmax=0,tmin=9999999,tavg=0;
   for (i=0;i<num_trips;i++) {
      source = trips[i].source_station;
      dest   = trips[i].dest_station;
      time   = trips[i].time_minutes;
      num_bikes[source]--;
      num_bikes[dest]++;
      tavg += time;
      if (time>tmax) {
         tmax=time; srcmax=source; destmax=dest;
      }
      if (time<tmin) {
         tmin=time; srcmin=source; destmin=dest;
      }
   }
   tavg /= num_trips;
   printf("Average time of trips: %.2f minutes\n",tavg);
   printf("Longest trip (%.2f min.) station %d to %d\n",tmax,srcmax,destmax);
   printf("Shortest trip (%.2f min.) station %d to %d\n",tmin,srcmin,destmin);
}
```

Parallelize the function by means of OpenMP in the most efficient way.