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];
}
```

Solution: There is a data dependency among the different iterations: violates the 1st Bernstein condition ($I_j \cap O_i \neq \emptyset$), since, for instance, x[2] is an output variable of iteration i=1 and an input variable of iteration i=2. It is not possible to eliminate this data dependency.

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

Solution: There is a data dependency among the different iterations: violates the 3rd Bernstein condition ($O_i \cap O_j \neq \emptyset$), since `x` is an output variable in every iteration. In this case it is indeed possible to remove the dependency:

```
for (i=0;i<N;i++) {
  a[i] = a[i] + y[i];
}
x = a[N-1];
```

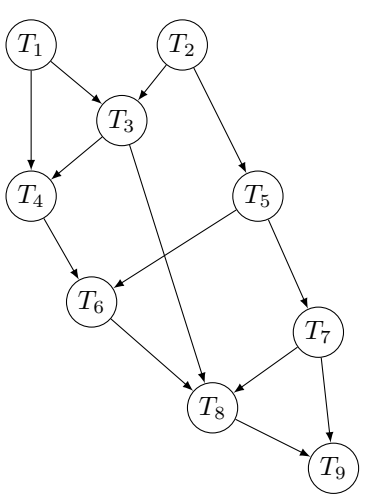
(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

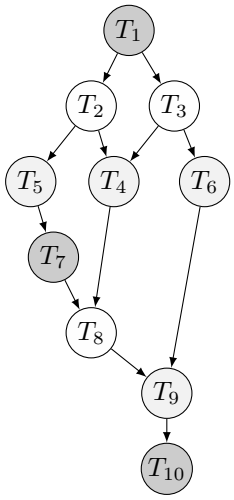
Solution: There is a data dependency among the different iterations: violates the 1st Bernstein condition ($I_j \cap O_i \neq \emptyset$), since, for instance, y[1] is an output variable of iteration i=1 and an input variable in iteration i=0. In this case it is indeed possible to remove the dependency:

```
x[N-2] = x[N-2] + y[N-1];
for (i=N-3;i>=0;i--) {
  y[i+1] = y[i+1] + z[i+1];
  x[i] = x[i] + y[i+1];
}
y[0] = y[0] + z[0];
```

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.

Solution: Among all possible paths between an initial node and a final node, the one that has the highest cost (critical path) is $T_1 - T_3 - T_4 - T_6 - T_8 - T_9$ (or equivalently beginning in T_2). Its length is $L = 6$. The average concurrency degree is

$$M = \sum_{i=1}^{9} \frac{1}{6} = \frac{9}{6} = 1.5$$

(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.

Solution: In this case, the critical path is $T_1 - T_2 - T_5 - T_7 - T_8 - T_9 - T_{10}$ and its length is

$$L = \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 = 7n^3 \text{ flops}$$

The average concurrency degree is

$$M = \frac{3 \cdot \frac{1}{3}n^3 + 3 \cdot n^3 + 4 \cdot 2n^3}{7n^3} = \frac{12n^3}{7n^3} = 1.71$$

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.

Solution:

(a)
```
void prodmvp(double a[N], double c[N], double B[N][N])
{
  int i, j;
  double sum;
  #pragma omp parallel for private(j,sum)
  for (i=0; i<N; i++) {
    sum = 0.0;
    for (j=0; j<N; j++)
      sum += B[i][j] * c[j];
    a[i] = sum;
  }
}
```

(b) Sequential cost: $t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 2 = 2N^2$ flops

Parallel cost: $t(N,p) = \sum_{i=0}^{d-1} \sum_{j=0}^{N-1} 2 = 2dN$ flops, where $d = \frac{N}{p}$

(c) Speedup: $S(N,p) = \frac{t(N)}{t(N,p)} = \frac{2N^2}{2dN} = \frac{N}{d} = p$

Efficiency: $E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1$

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).

Solution: The computations have two stages. During the first stage, each row in the matrix is overwritten with the next row multiplied by 2. The second stage computes the sum of all the elements from the matrix.

The first stage has only one operation in its inner loop, and therefore its cost is $\sum_{i=0}^{M-2}\sum_{j=0}^{N-1} 1 = \sum_{i=0}^{M-2} N = (M-1)N \approx MN$. [This is the asymptotic approximation, that is, assuming that both N and M are large.] Second stage has a similar cost, except for the `i` loop, which has one iteration more.

Total Sequential cost: $t_1 = 2MN$ flops

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

Solution: A parallelisation can be performed using two parallel regions, one per stage, since the second stage cannot start before the first one has ended. In the first stage, there are dependencies among the iterations of the `i`-loop, which can be solved by exchanging the loops and parallelising at the level of loop `j` (it would be also possible to parallelize loop `j` without exchanging the loops, but this would be less efficient). The second stage has no dependencies but requires a reduction on the variable `sum`. In both cases, loop counters `i` and `j` must be private variables.

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

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

Solution: In parallel the cost for the first stage is $\frac{N}{p}M$ and for the second stage the cost is $\frac{M}{p}N$ (in the latter, we neglected the cost associated to the reduction performed internally by OpenMP on the variable `sum`.)

The parallel time will be: $t_p = \dfrac{2MN}{p}$ flops

Therefore, the speed-up will be: $S_p = \dfrac{t_1}{t_p} = \dfrac{2MN}{2MN/p} = p$

(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.

Solution: In this case, the parallel time is $t_p = MN + \dfrac{MN}{p}$ flops, and therefore the speedup is

$$S_p = \frac{2MN}{MN + MN/p} = \frac{2}{1 + 1/p} = \frac{2p}{p+1},$$

whose limit when $p \to \infty$ is 2. That is, the speedup will never be greater that 2 even if we used many processors.

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.

Solution: First loop:

```
#pragma omp parallel for reduction(+:s) private(j,k,aux)
```

Second loop:

```
#pragma omp parallel for reduction(+:s) private(k,aux)
```

Third loop:

```
#pragma omp parallel for reduction(+:aux)
```

The most efficient approach consists in parallelizing the outer loop, since a lower overhead is produced due to the activation and deactivation of the threads, reducing also the waiting times due to the implicit synchronization at the end of the directive.

(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.

Solution: Sequential cost: $t_1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \left(1 + \sum_{k=0}^{n-1} 2 \right) \approx 2n^3$ flops.
Parallel cost: $t_n = 2n^3/n = 2n^2$ flops.
Speed-up: $S_n = t_1/t_n = n$.

(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.

Solution: Two integer variables are declared, `iter` and `tid` (`iter` must be initialized to 0 and `tid` must appear as private in the `parallel` clause, with `private(tid)`), and the following lines will be added:
Before the first and the second `for`:

```
tid = omp_get_thread_num();
if (!tid) iter++;
```

After the nested loops (before `return`):

```
printf("Number of iterations performed by thread 0 = %d\n",iter);
```

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.

Solution:

```
#include <stdio.h>
#include <omp.h>

int main() {
  int i, tid;
  unsigned int n, s=0;
  scanf("%u",&n);
  omp_set_num_threads(6);
  #pragma omp parallel for lastprivate(tid) reduction(+:s) schedule(dynamic,2)
  for (i=1;i<=n;i++) {
    tid = omp_get_thread_num();
    s+=i;
  }
  printf("The thread that summed the last number is %d\n",tid);
  printf("The sum of the first %u natural numbers is %u\n",n,s);
}
```

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.

Solution:

```
#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;
     #pragma omp parallel for private(j) reduction(+:err)
     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;
     }
```

(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.

Solution:

$$t(n) = \sum_{i=0}^{n-1}(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3) = \sum_{i=0}^{n-1}(2n+1) \approx 2n^2$$

$$t(n,p) = \sum_{i=0}^{\frac{n}{p}-1}(\sum_{j=0}^{i-1} 2 + \sum_{j=i+1}^{n-1} 2 + 3) = \sum_{i=0}^{\frac{n}{p}-1}(2n+1) \approx \frac{2n^2}{p}$$

$$S(n,p) = \frac{2n^2}{\frac{2n^2}{p}} = p$$

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.

Solution: The outermost loop cannot be parallelized since there is a dependency of each iteration with respect to the previous one, due to setting `x` with the value obtained for `x2` in the previous

iteration. In the `i` loop, variable `s` accumulates the value of the product of the row by the vector but this is computed completely in each iteration, and therefore it must be private (in the same way as the variable of the inner loop, `j`). This is not the case of variable `q`, whose value is the result of multiplying the values of the different iterations, requiring a reduction. Due to a dependency between the two `for` loops of `i`, it is not necessary to use a single parallel region since we cannot use the `nowait` clause. The variable `max` does not need protection because there are no race conditions, since all the iterations of the outer loop are sequential.

```
#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;
    #pragma omp parallel for private(s,j) reduction(*:q)
    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;
    }

    #pragma omp parallel for
    for (i=0;i<n;i++)
      x[i] = x2[i];

    if (max<q)
      max = q;
  }
  return max;
}
```

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

Solution:

$$t(n) = \sum_{i=0}^{n-1} \big(\sum_{j=0}^{n-1} 2 + 1 \big) = \sum_{i=0}^{n-1} (2n+1) = 2n^2 + n \approx 2n^2 \text{ flops}$$

(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.

Solution:

$$t(n,p) = \sum_{i=0}^{\frac{n}{p}-1} \big(\sum_{j=0}^{n-1} 2 + 1 \big) = 2\frac{n^2}{p} + \frac{n}{p} \approx 2\frac{n^2}{p}$$

$$S(n,p) = \frac{2n^2}{2\frac{n^2}{p}} = p$$

$$E(n,p) = 1$$

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.

Solution: This can be achieved by simply adding the following directive just before the loop:

```
#pragma omp parallel for private(j,k,mf,val)
```

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

Solution: This can be achieved by simply adding the following directive just before the loop:

```
#pragma omp parallel for private(k,val) reduction(min:mf)
```

(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.

Solution: The execution time for a single iteration of the `i` loop is:

$$1 + \sum_{j=0}^{N-1} \left(2 + \sum_{k=0}^{i-1} 2 \right) \approxeq \sum_{j=0}^{N-1} 2i = 2Ni \,\text{flops}$$

and the execution time of the entire function is the sum of the execution time of all the iterations of the loop, that is:

$$\sum_{i=0}^{M-1} 2Ni = 2N \sum_{i=0}^{M-1} i \approxeq NM^2 \,\text{flops}$$

(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.

Solution: In the previous part, we can see that the cost of each iteration of the `i` loop increases as `i` gets higher. If we use the `schedule(static)` scheduling we will obtain an imbalanced distribution

of the load, as the iterations with the highest cost (the largest values of `i`) will be allocated to the same thread.

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.

Solution:

The best approach will be to parallelise the outermost loop. Just add the following directive right before the `i` loop.

```
#pragma omp parallel for private(j, k, aux) reduction(+: s)
```

The iterations of the loop `i` have different costs, as the number of iterations in loop `k` depend on the value of `i`. Therefore, we could expect that the scheduling will affect the execution time. Since two consecutive iterations of the loop `i` will differ on 1 iteration in the loop `k`, a `static` or `dynamic` scheduling with a *chunk* size equal to 1 (e.g. `schedule(static,1)` or `schedule(dynamic,1)`) could be most adequate ones.

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

Solution:

$$t(N) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left(2+\sum_{k=i}^{N-1}2\right) \cong 2\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(N-i) = 2\sum_{i=0}^{N-1}\left(N^2-iN\right)$$
$$\cong 2\left(N^3-\frac{N^3}{2}\right) \cong N^3 \text{ flops}$$

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.

Solution: Right before the `i` loop we add the directive:

```
#pragma omp parallel for private(aux,j,x,y) reduction(+:stot)
```

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

Solution:

```
...
  aux = 0;
  #pragma omp parallel
  {
    #pragma omp for private(x) reduction(+:aux) nowait
    for (j=0; j<N; j++) {
      ...
    }
    #pragma omp for private(y)
    for (j=i; j<N; j++) {
      ...
    }
  } /* end of parallel */
  vs[i] = aux;
...
```

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

Solution:

$$t(N) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} 3 + \sum_{j=i}^{N-1} 1 + 1 \right) \approx \sum_{i=0}^{N-1} (3N + N - i) = 4N^2 - \sum_{i=0}^{N-1} i \approx 4N^2 - \frac{N^2}{2} = \frac{7N^2}{2} \text{ flops}$$

(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.

Solution:

$$t(N,p) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{\frac{N}{p}-1} 3 + \sum_{j=i}^{N-1} 1 + 1 \right) \approx \sum_{i=0}^{N-1} \left(\frac{3N}{p} + N - i \right) = \frac{3N^2}{p} + N^2 - \frac{N^2}{2} = \frac{3N^2}{p} + \frac{N^2}{2} \text{ flops}$$

If $p = N$:

$$t(N,p) = 3N + \frac{N^2}{2} \approx \frac{N^2}{2} \text{ flops}$$

Therefore the speed-up and efficiency are:

$$S(N,p) = \frac{\frac{7N^2}{2}}{\frac{N^2}{2}} = 7; \quad E(N,p) = \frac{7}{N}$$

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;
}
```

Solution: The variable `found` is declared as `volatile` in order to guarantee that as soon as one of the threads modifies its value the rest of threads see that change.

```
int search(int x[], int n, int value)
{
   volatile int found=0;
   int i, jump;
   #pragma omp parallel private(i)
   {
      i = omp_get_thread_num();
      jump = omp_get_num_threads();
      while (!found && i<n) {
         if (x[i]==value) found=1;
         i += jump;
      }
   }
   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`.

Solution:

```
double norm(double v[], int n)
{
   int i, i_thread, p;
   double r=0;

   /* Stage 1 */
   #pragma omp parallel private(i_thread)
   {
      p = omp_get_num_threads();
      i_thread = omp_get_thread_num();
      sums[i_thread]=0;
      #pragma omp for schedule(static)
      for (i=0; i<n; i++)
         sums[i_thread] += v[i]*v[i];
   }

   /* Stage 2 */
   for (i=0; i<p; i++)
      r += sums[i];
   return sqrt(r);
}
```

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

Solution:

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

(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.

Solution: The cost of the sequential algorithm (note that the cost of computing the square root is negligible with respect to the cost of loop `i`) is:

$$t(n) = \sum_{i=0}^{n-1} 2 \approx 2n \text{ flops}$$

The cost of the parallel algorithm: each iteration of the `i`-loop requires 2 flops and each thread computes n/p iterations. Then, the cost is $t(n,p) = 2n/p$ flops. The speed-up is $S(n,p) = 2n/(2n/p) = p$.

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.

Solution:

```
void f(int n, double a[], double b[])
{
   int i, cont;
   #pragma omp parallel private(cont)
   {
      cont=0;
      #pragma omp for
      for (i=0; i<n; i++) {
         b[i]=cos(a[i]);
         cont++;
      }
      printf("Thread %d: %d processed iterations\n",
                omp_get_thread_num(), cont);
```

```
    }
  }
```

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).

Solution:

The computation has two stages. The first stage sums all the squares of the elements of the matrix. The second stage divides each element by this previous sum. Using two separated parallel blocks we guarantee that the second stage does not start before the ending of the first stage, which will lead to an incorrect result. The first stage requires a reudction on the variable `sum`. In both cases the variable `j` must be `private`. The variable `factor` is `shared` and it is computed by the main thread (out of the parallel regions).

```
void normalize(double A[N][N])
{
  int i,j;
  double sum=0.0,factor;
  #pragma omp parallel for reduction(+:sum) private(j)
  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);
  #pragma omp parallel for private(j)
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] = factor*A[i][j];
    }
  }
}
```

(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.

Solution:

```
void normalize(double A[N][N])
{
  int i,j;
  double sum=0.0,factor;
  #pragma omp parallel private(j)
  {
    #pragma omp for reduction(+:sum)
    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);
    #pragma omp for
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        A[i][j] = factor*A[i][j];
      }
    }
  }
}
```

The clause `nowait` is used to avoid the implicit barrier at the end of a parallel block, such as the `for` directive. However, in this case we cannot use it as the barrier is needed to ensure that the value of `sum` is correct. If a thread starts executing the next `for` before other threads are still in the first one, the value of `sum` will be incorrect.

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.

Solution:

```
double ejp(double x[M], double y[N], double A[M][N])
{
   int i, j;
   double aux,s=0.0;
   #pragma omp parallel
   {
      #pragma omp for nowait
      for (i=0; i<M; i++)
         x[i] = x[i]*x[i];
      #pragma omp for
      for (i=0; i<N; i++)
         y[i] = 1.0+y[i];
      #pragma omp for private(j,aux) reduction(+:s)
      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;
}
```

Since the first two `for` loops are not inter-dependent, the clause `nowait` should be used at the end of the first `for` loop, to remove the automatic barrier at the end of the first `for` loop.

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

Solution:
Sequential time:

$$t(M,N) = M + N + 2MN \approx 2MN \text{ flops},$$

assuming that M and N are large enough (asymptotic cost).
Parallel time:

$$t(M,N,p) = \frac{M}{p} + \frac{N}{p} + \frac{2MN}{p} = \frac{M+N+2MN}{p} \approx \frac{2MN}{p} \text{ flops},$$

assuming that M and N are large enough (asymptotic cost).

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

Solution: Speed-up:

$$S(M,N,p) = \frac{t(M,N)}{t(M,N,p)} \approx \frac{2MN}{\frac{2MN}{p}} = p$$

Efficiency:

$$E(M,N,p) = \frac{S(M,N,p)}{p} = 1$$

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;
```

Solution: The only thing to take into account is that variable `a` must be private.

```
#pragma omp parallel sections private(a)
{
  #pragma omp section
  {
    a = -1.8;
    fun1(n,&a);
    b[0] = a;
  }
  #pragma omp section
  {
    a = 3.2;
    fun2(n,&a);
    b[1] = a;
  }
  #pragma omp section
  {
    a = 0.25;
    fun3(n,&a);
    b[2] = a;
  }
}
```

Question 2–7

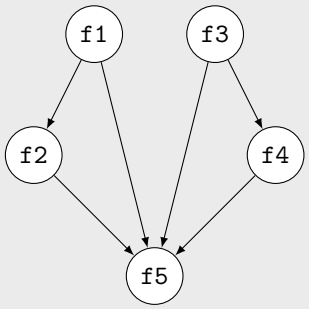
Given the following function:

```
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.

Solution: The dependency graph is shown below:



The dependencies of `f5` with respect to the other tasks are flow dependencies (their inputs are generated by other tasks). The dependencies of `f2` with `f1` and `f4` with `f3` are anti-dependencies (it is not that they need the ouput of other tasks, but they modify the input of previous tasks and therefore they must not be executed until these previous tasks have finished). There is also an output dependency between `f5` and `f1` because both of them modify the same output data (vector `a`).

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

Solution:

```
void func(double a[],double b[],double c[],double d[])
{
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      f1(a,b);
      f2(b,b);
    }
    #pragma omp section
    {
      f3(c,d);
      f4(d,d);
    }
  }
  f5(a,a,b,c,d);
}
```

(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?

Solution: For convenience, we assume that each function takes 1 unit of time. The sequential time is

$$t_1 = 1 + 1 + 1 + 1 + 1 = 5$$

In order to obtain a high speedup we must reduce the execution time as much as possible. This leads to use as many processors as parallel tasks can be executed concurrently. Given the parallel

program and the dependency graph, it is enough with 2 processors (threads). In this case, the cost would be (`f1` and `f2` will be done in parallel to `f3` and `f4`):

$$t_p = 2 + 1 = 3$$

$$Sp = t_1/t_p = 5/3 = 1.67$$

In order to improve the speedup, it would be possible to eliminate the anti-dependencies by replicating the data that are going to be modified. It remains to see if the cost of the copy and the required extra storage compensates the obtained speedup. By doing this, `b` and `d` would be copied before starting to work. `f1` and `f3` would work with the copies and `f2` and `f4` with the real data. With this `f1`,`f2`,`f3`,`f4` could be run at the same time, so that we have (without taking into account the overhead of the copies):

$$t_p = 1 + 1 = 2 \qquad \text{(working with 4 processors)}$$

$$S_p = 5/2 = 2.5$$

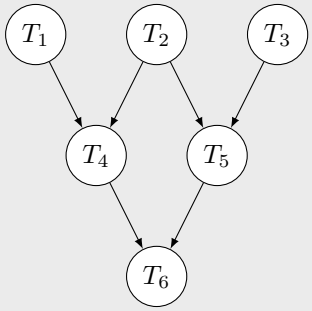
Question 2–8

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.

Solution:



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

Solution:

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

   #pragma omp parallel private(i,j)
   {
      #pragma omp sections
      {
         #pragma omp section
         T1(x,n);    /* Task T1 */
         #pragma omp section
         T2(y,n);    /* Task T2 */
         #pragma omp section
         T3(z,n);    /* Task T3 */
      }

      #pragma omp sections
      {
         #pragma omp section
         /* 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;
         }

         #pragma omp section
         /* 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;
    }
```

(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.

Solution: The a priori cost of T_4 is:

$$\sum_{i=0}^{n-1}\left(\sum_{j=0}^{n-1}2+\sum_{j=0}^{n-1}1\right)=\sum_{i=0}^{n-1}(2n+n)=3n^2 \text{ flops}$$

The cost of T_5 is equal to that of T_4, and the cost of T_6 is $2n+1$ flops.
Therefore, the sequential cost is:

$$t(n)=2n^2+2n^2+2n^2+3n^2+3n^2+2n+1\approx 12n^2 \text{ flops}$$

If the number of threads, p, is at least 3, the cost of the parallel algorithm will be $t(n,p)=2n^2+3n^2+2n\approx 5n^2$ flops, and the speedup will be:

$$S(n,p)=\frac{12n^2}{5n^2}=2.4$$

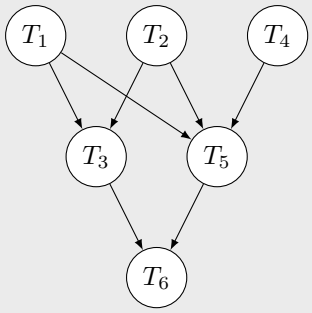
Question 2–9

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.

Solution: Task T_5 updates `x`, so it has a reverse-dependency with respect to T_1, T_2 and T_4.



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

Solution:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      minx = minimum(x,n);       /* T1 */
      #pragma omp section
      maxx = maximum(x,n);       /* T2 */
      #pragma omp section
      compute_y(y,x,n);         /* T4 */
   }
   #pragma omp sections
   {
      #pragma omp section
      compute_z(z,minx,maxx,n); /* T3 */
      #pragma omp section
      compute_x(x,y,n);         /* T5 */
   }

}
compute_v(v,z,x);          /* T6 */
```

(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.

Solution: Length of the critical path: $L = 2n + n + n = 4n$ flops

Average concurrency degree: $\frac{7n}{4n} = 7/4 = 1.75$

For obtaining the speed-up and the efficiency, we first have to compute the sequential execution time and the parallel execution time when 5 processores are used.

$$t(n) = 7n$$

$$t(n, 5) = 2n + n + n = 4n$$

Therefore

$$S(n, p) = \frac{7n}{4n} = 1.75$$

$$E(n, p) = \frac{1.75}{5} = 0.35$$

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);
```

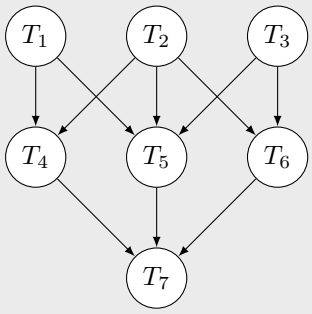
(a) Parallelize the code efficiently at the level of the loops.

Solution: The loop in `fun1` cannot be parallelized due to the dependencies between the different iterations, and therefore we only consider the function `compare`.

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

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

Solution:



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

Solution: The parallel code of the main program is the following (the rest stays invariant):

```
int i, n=10;
```

```
double a[10], b[10], c[10], x=5, y=7, z=11, w;

#pragma omp parallel sections
{
  #pragma omp section
  fun1(a,n,x);
  #pragma omp section
  fun1(b,n,y);
  #pragma omp section
  fun1(c,n,z);
}
#pragma omp parallel sections
{
  #pragma omp section
  x = compare(a,b,n);
  #pragma omp section
  y = compare(a,c,n);
  #pragma omp section
  z = compare(c,b,n);
}
w = x+y+z;
printf("w:%f\n", w);
```

(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.

Solution: The sequential time is:

$$t(n) = 3\sum_{i=1}^{n-1} 1 + 3\sum_{i=0}^{n-1} 3 + 2 \approx 3n + 9n = 12n$$

The parallel time and speed-up for the first algorithm considering 3 processors are:

$$t_1(n,3) = 3\sum_{i=1}^{n-1} 1 + 3\sum_{i=0}^{\frac{n}{3}-1} 3 + 2 \approx 3n + 3n = 6n$$

$$S_1(n,3) = \frac{12n}{6n} = 2$$

The parallel time and speed-up for the second algorithm considering 3 processors are:

$$t_2(n,3) = \sum_{i=1}^{n-1} 1 + \sum_{i=0}^{n-1} 3 + 2 \approx n + 3n = 4n$$

$$S_2(n,3) = \frac{12n}{4n} = 3$$

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);
```

Solution: Parallelization of loops is not possible in this case because the number of iterations is not known a priori. Therefore, we do a task parallelization, placing each loop (including its initialization) in a concurrent section. In order for the result to be correct, we must also take into account that variables `x`, `y`, `z` must have a private scope, since they are modified in the three loops but their value is not shared among the loops (they are initialized at the beginning of each loop). Finally, variable `w` must have a final value equal to the accumulated value of all concurrent sections, so we use a `reduction` clause for this.

```
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 */

#pragma omp parallel sections private(x,y,z) reduction(+:w)
{
  #pragma omp section
  {
    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);
  }
  #pragma omp section
  {
    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);
  }
  #pragma omp section
  {
    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.

Solution: In both functions, the most efficient way of doing the loop parallelization is to parallelize the outer loop by means of the directive `parallel for`. In this way, the iterations of that loop are distributed among the different threads in such a way that each thread is in charge of the computation associated with a certain subset of the rows of the result matrix. Additionally, we must define the scope of the variables, so that `j`, `k` and `sum` must be private for each thread.

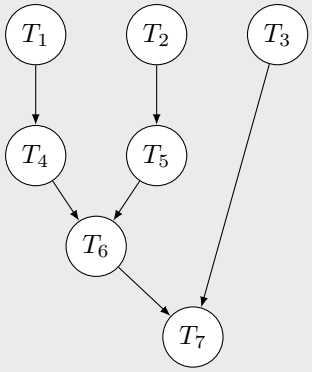
```
void matmult(double A[N][N],
       double B[N][N],double C[N][N])
{
  int i,j,k;
  double sum;
  #pragma omp parallel for private(j,k,sum)
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum = 0.0;
      for (k=0; k<N; k++) {
        sum = sum + A[i][k]*B[k][j];
      }
      C[i][j] = sum;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

```
void symmetrize(double A[N][N])
{
  int i,j;
  double sum;
  #pragma omp parallel for private(j,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;
    }
  }
}
```

(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.

Solution: The task dependency graph is:



The maximum degree of concurrency is 3, since there could not be more than 3 tasks concurrently running.

The cost in flops of `matmult` (c_m) and of `symmetrize` (c_s) is:

$$c_m = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1} 2 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2N = \sum_{i=0}^{N-1} 2N^2 = 2N^3.$$

$$c_s = \sum_{i=0}^{N-1}\sum_{j=0}^{i} 1 = \sum_{i=0}^{N-1}(i+1) \approx \frac{N^2}{2}.$$

Therefore, five of the tasks have a cost on the order $2N^3$, whereas T_4 and T_5 have a lower cost ($N^2/2$). The total accumulated cost for all the tasks is $C = 10N^3 + N^2$ (sequential cost).

The critical path is T_1–T_4–T_6–T_7 (or, equivalently, T_2–T_5–T_6–T_7), whose cost is

$$L = 2N^3 + \frac{N^2}{2} + 2N^3 + 2N^3 = 6N^3 + \frac{N^2}{2}.$$

the average degree of concurrency is

$$M = \frac{C}{L} = \frac{10N^3 + N^2}{6N^3 + N^2/2} \approx \frac{10}{6} = 1,67.$$

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

Solution:

We can group task T_1 with T_4 and T_2 with T_5, since there are not cross-dependencies among both groups. Then, a possible (among others) solution could be to implement a first part with three sections, followed by the parallel execution of T_3 and T_6, and ending with the execution of T_7, once the other sections have finished:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
```

```
          matmult(X,Y,C1);
          symmetrize(C1);
        }
        #pragma omp section
        {
          matmult(Y,Z,C2);
          symmetrize(C2);
        }
      }
      #pragma omp sections
      {
        #pragma omp section
        matmult(Z,X,C3);
        #pragma omp section
        matmult(C1,C2,D1);
      }
    }
    matmult(D1,C3,D);
```

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.

Solution:

$$t_1 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 + \sum_{i=1}^{N-1} 1 + \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} 1 = N^2 + N - 1 + N^2 = 2N^2 + N - 1 \approx 2N^2$$

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

Solution: The loop that computes the prefix sum cannot be parallelised, as there are data dependencies among the iterations. Despite there are (rather complex) algorithmic schemas that can be applied to compute the prefix sum in parallel, we have decided to implement this stage sequentially, as it has a linear cost, whereas the other two parts have a quadratic costs. Therefore, the penalty for not parallelising the prefix sum is negligible for large values of de problem size.

In order to perform the prefix sum sequentially within a parallel region, we should include the `single` directive. Otherwise, we may face a race condition in this part.

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

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

Solution: The parallel time will be:

$$t_p = \sum_{i=0}^{N/p-1} \sum_{j=0}^{N-1} 1 + \sum_{i=1}^{N-1} 1 + \sum_{j=0}^{N/p-1} \sum_{i=0}^{N-1} 1 = \frac{N^2}{p} + N - 1 + \frac{N^2}{p} = \frac{2N^2}{p} + N - 1 \approx \frac{2N^2}{p}$$

Therefore, the speed-up will be:

$$S_p = \frac{t_1}{t_p} \approx \frac{2N^2}{2N^2/p} = p$$

A linear speed-up is achieved, although the real value could be smaller if we would have kept the lower-order terms on the expressions for t_1 and t_p.

Question 2–14

Given the following function:

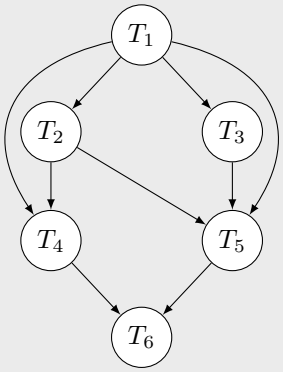
```
double calcula()
{
  double A[N][N],B[N][N],a,b,x,y,z;

  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.

Solution: The dependency graph will be:



The maximum concurrency degree is 2 (for example, T_2 and T_3 can be executed simultaneously). Considering the cost for each task, the critical path can go through either T_2 or T_3 as both have the same cost. It must go through T_5, though, as T_5 has a higher cost than T_4. Therefore, there are two equivalent critical path: T_1, T_2, T_5, T_6 and T_1, T_3, T_5, T_6. Both have the same length:

$$L = 2n^2 + n^2 + 2n^2 + 1 = 5n^2 + 1 \quad \text{flops}$$

The average concurrency degree will be:

$$M = \frac{\sum_{i=1}^{6} c_i}{L} = \frac{7n^2 + 1}{5n^2 + 1} \approx 1.4$$

(b) Parallelise the function using OpenMP.

Solution:

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

  rellena(A,B);               /* T1 */
  #pragma omp parallel sections
  {
    #pragma omp section
    a = calculos(A);          /* T2 */
    #pragma omp section
```

```
      b = calculos(B);          /* T3 */
    }
    #pragma omp parallel sections
    {
      #pragma omp section
      x = suma_menores(B,a);    /* T4 */
      #pragma omp section
      y = suma_en_rango(B,a,b); /* T5 */
    }
    z = x + y;                  /* T6 */

    return z;
  }
```

(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.

Solution: As there are two `section` in each one of the `sections` of the code, we can execute them in parallel if 2 or more threads are used. Therefore, the parallel execution time can be obtained by computing the maximum time of the different tasks appearing in each one of the `sections`.

$$t_1 = 2n^2 + n^2 + n^2 + n^2 + 2n^2 + 1 = 7n^2 + 1 \quad \text{flops}$$

$$t_3 = 2n^2 + n^2 + 2n^2 + 1 = 5n^2 + 1 \quad \text{flops}$$

$$S_p = \frac{t_1}{t_3} \approx 1.4$$

$$E = \frac{Sp}{p} = \frac{Sp}{3} = 46.67\%$$

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.

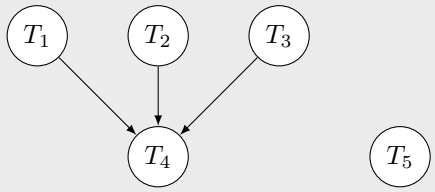
```
typedef double image[N][M];

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).

Solution: The concurrency degree is shown next:



The maximum concurrency degree is 4, as T_1, T_2, T_3 and T_5 can be executed in parallel. To obtain the average concurrency degree, we must determine the cost in flops of functions `difer` and `sum`:

$$\text{Cost of } \texttt{difer} = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} 1 = N \cdot M \qquad \text{Cost of } \texttt{sum} = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} 2 = 2 \cdot N \cdot M$$

The critical path is $T_1 \longrightarrow T_4$ (or any of the other two paths in the graphs that end up in T_4, as they have the same cost), so the length of the critical path is:

$$L = N \cdot M + 2 \cdot N \cdot M = 3 \cdot N \cdot M$$

Therefore,

$$\text{Average concurrency degree} = \frac{N \cdot M + N \cdot M + N \cdot M + 2 \cdot N \cdot M + N \cdot M}{L} = \frac{6}{3} = 2$$

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

Solution: We implement the parallelisation by sections. Task T_5 can be executed in parallel along with any of the other tasks. In our parallel code we have placed with T_4, as this will be the solution with higher concurrency in the case of using 3 threads.

```
void procesa(image f1,image f2,image f3,image f4,image r1,image r2)
{
  image d1,d2,d3;
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      difer(f2,f1,d1);          /* T1 */
      #pragma omp section
      difer(f3,f2,d2);          /* T2 */
      #pragma omp section
      difer(f4,f3,d3);          /* T3 */
    }
    #pragma omp sections
    {
```

```
          #pragma omp section
          sum(d1,d2,d3,r1);        /* T4 */
          #pragma omp section
          difer(f4,f1,r2);          /* T5 */
        }
      }
    }
```

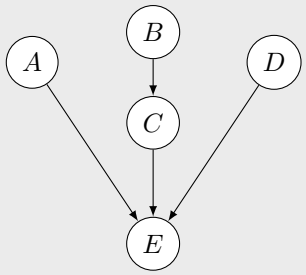
Question 2–16

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.

Solution:



Maximum concurrency degree: 3 (for example, tasks A, B and D can be performed simultaneously).
The critical path is $A \rightarrow E$, whose length is $3n^2 + 2$ flops.
The average concurrency degree is $\frac{3n^2+n^2+n^2+2n^2+2}{L} = \frac{7n^2+2}{3n^2+2} \approx 7/3 = 2.33$.

(b) Implement a parallel version using OpenMP.

Solution:

```
double matrix_computations(double mat[n][n])
{
  double x,y,z,aux,total;
  #pragma omp parallel sections
  {
    #pragma omp section
    x = A(mat);
    #pragma omp section
    {
      aux = B(mat);
      y = C(mat,aux);
```

```
    }
    #pragma omp section
    z = D(mat);
  }
  total = x + y + z;
  return total;
}
```

(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.

Solution: $t_1 = 3n^2 + n^2 + n^2 + 2n^2 + 2 = 7n^2 + 2$ flops

If we execute the parallel code with 2 threads, the best load balancing is achieved if a thread runs task A meanwhile the other thread executes the rest of the tasks: $t_2 = \max(3n^2, n^2 + n^2 + 2n^2) + 2 = \max(3n^2, 4n^2) + 2 = 4n^2 + 2$ flops

$S_2 = \frac{t_1}{t_2} = \frac{7n^2+2}{4n^2+2} \approx 7/4 = 1.75$

$E_2 = \frac{S_2}{2} = \frac{1.75}{2} = 87.5\%$

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

Solution:

```
#include <omp.h>
double matrix_computations(double mat[n][n])
{
  double x,y,z,aux,total,t1,t2,t;
  int num_threads;
  t1 = omp_get_wtime();
  #pragma omp parallel sections
  {
    #pragma omp section
    x = A(mat);
    #pragma omp section
    {
      aux = B(mat);
      y = C(mat,aux);
    }
    #pragma omp section
    z = D(mat);
  }
  total = x + y + z;
  t2 = omp_get_wtime();
  t = t2 - t1;
  #pragma omp parallel
  num_threads = omp_get_num_threads();
  printf("Time with %d threads: %.2f seconds\n",num_threads,t);
  return total;
}
```

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.

Solution:
Simply place the next directive before the `i` loop:

```
#pragma omp parallel for private(y,j,j2,a) reduction(+:x)
```

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

Solution:

```
...
for (i=0; i<M; i++) {
   y = 1;
   #pragma omp parallel
   {
      #pragma omp for private(a) reduction(+:x) nowait
      for (j=0; j<N; j++) {
         ...
      }
      #pragma omp for reduction(*:y)
      for (j2=1; j2<i; j2++) {
         ...
      }
   }
   pf[i] = y;
}
...
```

(c) Compute the sequential cost:

Solution:

$$\sum_{i=0}^{M-1}\left(\sum_{j=0}^{N-1}1+\sum_{j2=1}^{i-1}2\right)\approx\sum_{i=0}^{M-1}(N+2i)=\sum_{i=0}^{M-1}N+2\sum_{i=0}^{M-1}i\approx NM+M^2 \text{ flops}$$

(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.

Solution:

- Loop `i`: Different scheduling policies in loop `i` will lead to different costs, as the number of iterations of loop `j2` depend on the value of `i`. To balance the execution cost, the best scheduling policies will be *static* with a *chunk* of size 1, *dynamic* or *guided*.
- Loop `j`: Every iteration in the loop has the same cost (1 Flop). Therefore, the scheduling policies will not affect the performance (although dynamic scheduling may add an overhead).
- Loop `j2`: Same as in loop `j` (in this case, each iteration costs 2 Flops).

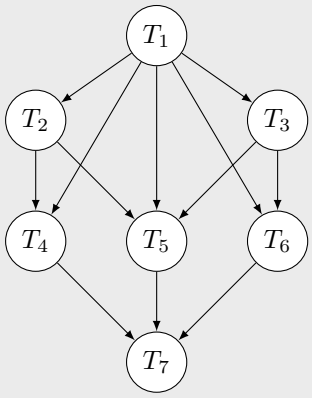
Question 2–18

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.

Solution:



(b) Parallelise the function efficiently using OpenMP.

Solution: The parallelised function will be:

```
void function() {
  double x,y,z,a,b,c,d,e;
  int n;
  n = read_data(&x,&y,&z);   /* Task 1 */
```

```
    #pragma omp parallel
    {
      #pragma omp sections
      {
        #pragma omp section
        a = f2(x,n);           /* Task 2 */
        #pragma omp section
        b = f3(y,n);           /* Task 3 */
      }
      #pragma omp sections
      {
        #pragma omp section
        c = f4(z,a,n);         /* Task 4 */
        #pragma omp section
        d = f5(&x,&y,n);       /* Task 5 */
        #pragma omp section
        e = f6(z,b,n);         /* Task 6 */
      }
    }
    write_results(c,d,e);      /* Task 7 */
  }
```

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

Solution: The sequential time is:

$$t(n) = 2n + 4n + 5n^2 = 5n^2 + 6n \simeq 5n^2$$

and the parallel time for 3 processing units:

$$t(n,p) = n + 2n + 3n^2 + n = 3n^2 + 4n \simeq 3n^2$$

Therefore, the speed-up will be:

$$S(n,p) = \frac{t(n)}{t(n,p)} \simeq \frac{5n^2}{3n^2} = 1,67$$

and the efficiency:

$$E(n,p) = \frac{S(n,p)}{p} \simeq \frac{1,67}{3} = 0,56$$

(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.

Solution: One of the critical paths is $T_1 - T_2 - T_5 - T_7$. The length of the critical path is:

$$L = n + 2n + 3n^2 + n = 3n^2 + 4n$$

From the length of the critical path, the average concurrency degree is:

$$M = \frac{\sum_{i=1}^{7} C_i}{L} = \frac{2n + 4n + 5n^2}{3n^2 + 4n} = \frac{5n^2 + 6n}{3n^2 + 4n} = \frac{5n + 6}{3n + 4} \simeq \frac{5n}{3n} = 1,67$$

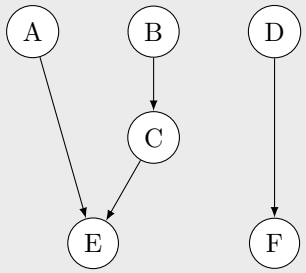
Question 2–19

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.

Solution:



Critical Path: A$\longrightarrow$E

Length of the critical path: $L = 2n^2 + n^2 = 3n^2$ flops

Average concurrency degree: $M = \frac{2n^2+n+n^2+2n^2+n^2+3n}{3n^2} \approxeq \frac{6n^2}{3n^2} = 2$

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

Solution:

```
double fpar(double x[], double y[], double z[], double v[], double w[]) {
   double r1, res;
   #pragma omp parallel
   {
      #pragma omp sections
      {
         #pragma omp section
         A(x,v);
         #pragma omp section
         { B(y,v,w); C(w,v); }
         #pragma omp section
         r1=D(z,v);
      }
      #pragma omp sections
      {
         #pragma omp section
         E(x,v,w);
         #pragma omp section
```

```
            res=F(z,r1);
          }
        }
        return res;
    }
```

(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.

Solution: The 3 sections in the first parallel block `sections` should be distributed between the 2 threads. The best case will be when tasks A and D are executed in different threads and tasks B and C in any of the two threads. For example, one thread could execute tasks A, B and C (with a cost of $2n^2 + n^2 + n \cong 3n^2$ flops) and the other thread will run task D ($2n^2$ flops). Then, one thread will run task E (n^2 flops) and the other will execute task F ($3n$ flops). Therefore, the cost will be:

$$t_2 = \max(3n^2, 2n^2) + \max(n^2, 3n) = 3n^2 + n^2 = 4n^2 \text{ flops}$$

Considering that the sequential cost is the sum of the cost of every task (that is $6n^2$ flops):

$$S_2 = \frac{6n^2}{4n^2} = \frac{6}{4} = 1.5$$

$$E_2 = \frac{1.5}{2} = 0.75$$

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.

Solution:

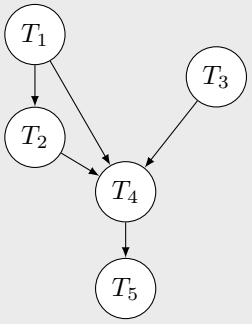
$$t_{\text{tarea4}} = 3$$

$$t_{\text{tarea5}}(n) = \sum_{i=0}^{n-1} 6 = 6n$$

$$t(n) = n + n + 4n + 3 + 6n \approx 12n \text{ flops}$$

(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.

Solution:



The maximum number of tasks that could run concurrently is 2 (tasks 1 and 3 or tasks 2 and 3), so the maximum concurrency degree is 2.

The critical path is the longest path between an initial and a final node. In this case, the critical path will be the one that goes through tasks 3, 4 and 5, with a length $L = 4n + 3 + 6n \approx 10n$

The average concurrency degree can be obtained as:

$$M = \frac{\sum_{i=1}^{6} t_{\text{tarea i}}}{L} = \frac{12n+3}{10n+3} \approx \frac{12}{10} = 1.2$$

(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.

Solution:

```
int exercise(double v[n],double x)
{
  int i,j,k=0;
  double a,b,c;
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        a = tarea1(v,x);
        b = tarea2(v,a);
      }
      #pragma omp section
      c = tarea3(v,x);
    }
    #pragma omp single
    x = x + a + b + c;   /* tarea 4 */
    #pragma omp for private(j) reduction(+:k)
    for (i=0; i<n; i++) {   /* tarea 5 */
      j = f(v[i],x);
      if (j>0 && j<4) k++;
    }
  }
  return k;
}
```

(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.

Solution:

$$t_6(n) = \max\{t_{\text{tarea1}} + t_{\text{tarea2}}, t_{\text{tarea3}}\} + t_{\text{tarea4}} + \sum_{i=0}^{n/6-1} 6 = 4n + 3 + \frac{6n}{6} \approx 5n \text{ flops}$$

$$S_6 = \frac{t(n)}{t_6(n)} = \frac{12n}{5n} = \frac{12}{5} = 2.4$$

$$E_6 = \frac{S_6}{p} = \frac{2.4}{6} = 0.4 \rightarrow 40\%$$

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;
    }
  }
}
```

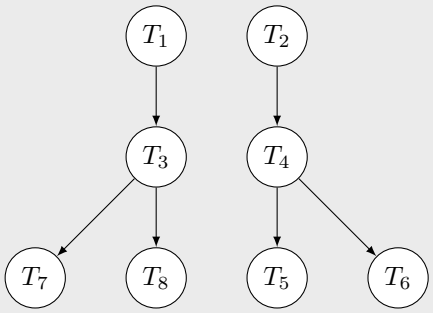
```
void normalize(double A[N][N]) {
  int i,j;
  double sum=0.0,factor;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      sum += A[i][j]*A[i][j];
    }
  }
  factor = 1.0/sqrt(sum);
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      A[i][j] *= factor;
    }
  }
}
```

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.

Solution: The task dependency graph is the following:



The cost in flops of `matmult` (c_m) and `normalize` (c_n) is:

$$c_m = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1} 2 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2N = \sum_{i=0}^{N-1} 2N^2 = 2N^3.$$

$$c_n = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 2 + 6 + \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 = 2N^2 + 6 + N^2 \approx 3N^2.$$

Therefore, six tasks have a cost of $2N^3$, whereas T_3 and T_4 have a much smaller cost ($3N^2$). The total accumulated cost is $C = 12N^3 + 6N^2$ (sequential cost).

The critical path is for instance T_1–T_3–T_8 (there are other paths with the same cost), whose cost is

$$L = 2N^3 + 3N^2 + 2N^3 = 4N^3 + 3N^2.$$

The average degree of concurrency is

$$M = \frac{C}{L} = \frac{12N^3 + 6N^2}{4N^3 + 3N^2} \approx \frac{12}{4} = 3.$$

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

Solution: We can group together task T_1 with T_3 and T_2 with T_4, since there are no cross-dependencies. Therefore, the parallelization can be done with a first part with two sections, and later execute the remaining four tasks in parallel.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      matmult(A,B,R1);    /* T1 */
      normalize(R1);      /* T3 */
    }
    #pragma omp section
    {
      matmult(C,D,R2);    /* T2 */
      normalize(R2);      /* T4 */
    }
  }
  #pragma omp sections
```

```
      {
        #pragma omp section
        matmult(A,R2,M1);   /* T5 */
        #pragma omp section
        matmult(B,R2,M2);   /* T6 */
        #pragma omp section
        matmult(C,R1,M3);   /* T7 */
        #pragma omp section
        matmult(D,R1,M4);   /* T8 */
      }
    }
```

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.

Solution:

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

       #pragma omp parallel for private(maximum,j) reduction(+:sum)
       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;
}
```

(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*.

Solution: The iterations of the i loop are distributed among the p available threads, and each iteration performs the inner loops completely, with a cost of

$$\sum_{i=0}^{N/p-1} \left(\sum_{j=0}^{M-1} 1 + \sum_{j=0}^{M-1} 3 \right) = \sum_{i=0}^{N/p-1} 4M = \frac{4NM}{p}$$

(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.

Solution:

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

   #pragma omp parallel private(maximum,j,count) reduction(+:sum)
   {
      count=0;
      #pragma omp for
      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];
               count++;
            }
         }
      }
      printf("Thread %d has summed %d elements.\n" ,
             omp_get_thread_num(), count);
   }
   return sum;
}
```

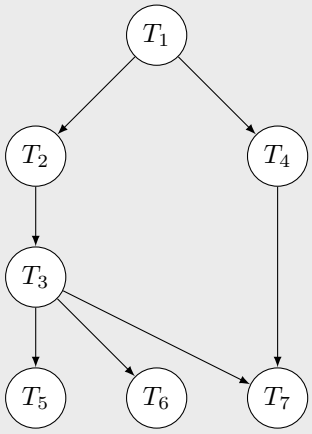
Question 2–23

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.

Solution: The task dependency graph is shown next.



Between T_2 and T_3 we have a flow dependency and an output dependency due to variable `c`. Between T_4 and T_7 there is an anti-dependency because of variable `b`.

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

Solution: From all possible paths, the one that corresponds to the critical path follows the sequence of tasks $T_1 - T_2 - T_3 - T_6$, with length L=10+15+8+35=68 flops.

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

Solution:

```
double a,b,c,e,d,f;
T1(&a,&b);
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
      c=T2(a);
      c=T3(c);
      }
      #pragma omp section
      d=T4(b);
   }
   #pragma omp sections
```

```
        {
           #pragma omp section
           e=T5(c);
           #pragma omp section
           f=T6(c);
           #pragma omp section
           b=T7(c);
        }
     }
```

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

Solution: The parallel cost can be obtained from the dependency graph, as the sum of the following costs: T_1, the maximum between $(T_2 + T_3)$ and T_4, and the maximum between T_5, T_6 and T_7. Therefore, $t_p = 10 + 23 + 35 = 68$.

We get the speedup as $S_p = \dfrac{t_1}{t_p} = \dfrac{148}{68} = 2.18$

Finally, the efficiency for 4 threads is computed as $E_p = \dfrac{S_p}{p} = \dfrac{2.18}{4} = 0.55$

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.

Solution: The parallelization of the previous code consists in parallelizing the inner loops. The outer `while` loop is not parallelizable, but the inner ones are if we take into account data dependencies. It boils down to simply add the following directive

```
#pragma omp parallel for private(a)
```

before each `for` loop. The variable `a` must be private, while the rest are shared except for the loop variable `i` which is private by default.

The variable `sorted` is shared and all threads access it for writing. This should oblige to protect it in a critical section (or a reduction operation). However, in this case it is not necessary.

(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.

Solution: The way of counting the number of exchanges consists in using a counter variable (`int cont=0`). This variable must be private to each thread, aggregating the total count at the exit of the loop by means of a reduction operation. The directive in the loops would be the following one:

```
#pragma omp parallel for private(a) reduction(+:cont)
```

adding in the body of the loop (inside the `if`) the instruction `cont++`.

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.

Solution: The usage of vector `ind` implies that several iterations of the loop may end up accessing the same element of vector `x`. For this reason, the computation of the maximum requires a critical section in order to avoid race conditions.

```
void f(int n, double v[], double x[], int ind[])
{
   int i;
   double aux;
   #pragma omp parallel for private(aux)
   for (i=0; i<n; i++) {
      aux=f2(v[i]);
      if (aux>x[ind[i]]) {
         #pragma omp critical
         if (aux>x[ind[i]]) {
            x[ind[i]] = aux;
         }
      }
```

```
      }
   }
```

The evaluation of function `f2` is stored in an auxiliary variable to avoid being done several times. Furthermore, it would be very inefficient to make the evaluation inside the critical section, as it would be done sequentially.

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.

Solution: In case of several occurrences, the sequential code returns the last position, which can be obtained in the parallel code by means of the maximum.

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

   #pragma omp parallel for reduction(max:pos)
   for (i=0; i<n; i++)
      if (x[i]==value)
         pos=i;

   return pos;
}
```

The previous solution does not work in OpenMP versions prior to 3.1. An alternative solution is:

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

   #pragma omp parallel for
   for (i=0; i<n; i++)
      if (x[i]==value)
         if (i>pos)
            #pragma omp critical
            if (i>pos)
               pos=i;
```

```
    return pos;
}
```

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.

Solution:

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

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

  #pragma omp parallel for private(j,s)
  for (i=0; i<n; i++) {
    s = 0;
    for (j=0; j<n; j++)
      s += fabs(A[i][j]);
    if (s>norm)
      #pragma omp critical
      if (s>norm)
        norm = s;
  }
  return norm;
}
```

The parallelisation is performed at the level of the outer loop to achieve a higher granularity and reduced synchronization cost. The parallelisation has a race condition in the update of the maximum of the sum of rows. To avoid errors produced by concurrent writes, a critical section is used to protect the concurrent writing on `norm`. To prevent an excessive sequentialisation, the critical section is included after the checking of the maximum, which requires an additional checking before the update of `norm`.

An alternative solution is to modify the `parallel` directive as shown below (and therefore it is not necessary to use the `critical` directive). However, this variant would not be valid in older versions of OpenMP (prior to 3.1), since reductions with the `max` operator were not allowed.

```
#pragma omp parallel for private(j,s) reduction(max:norm)
```

(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.

Solution: $t(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 2 = 2n^2$ flops $\qquad t(n,p) = \sum_{i=0}^{\frac{n}{p}-1} \sum_{j=0}^{n-1} 2 = 2\frac{n^2}{p}$ flops

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

Solution: $S(n,p) = \frac{t(n)}{t(n,p)} = p \qquad E(n,p) = \frac{S(n,p)}{p} = 1$
The speed-up is the maximum possible one, and the efficiency is in the order of 100%, which denotes an optimal utilization of the processors power.

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.

Solution:

```
double prod1(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel for reduction(*:p)
  for (i=0;i<n;i++)
    p *= v[i];
  return p;
}
```

(b) Without using reduction.

Solution: Here we present two different implementations, and comment which one is more efficient.

```
double prod2(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel for
  for (i=0;i<n;i++) {
    #pragma omp atomic
    p *= v[i];
  }
  return p;
}

double prod3(double v[], int n)
{
  double p=1;
  int i;
  #pragma omp parallel
  {
    int ppriv=1;   /* ppriv is a private variable */
    #pragma omp for nowait
    for (i=0;i<n;i++)
      ppriv *= v[i];
    #pragma omp atomic
    p *= ppriv;
  }
  return p;
}
```

The implementation `prod3` is more efficient than the implementation `prod2`, since the number of `atomic` operations is smaller.

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

(a) Using reduction.

(b) Without using reduction.

Solution:

(a)
```
int factorial1(int N)
{
  int fact = 1, n;
  #pragma omp parallel for reduction(*:fact)
  for (n=2; n<=N; n++)
    fact *= n;
  return fact;
}
```

(b) In this section we show two different implementation, discussing which one is most efficient.

```
int factorial2(int N)
{
  int fact = 1, n;
  #pragma omp parallel for
  for(n=2; n<=N; n++) {
    #pragma omp atomic
    fact *= n;
  }
  return fact;
}

int factorial3(int N)
{
  int fact = 1, n;
  #pragma omp parallel
  {
    int facp = 1; /* facp es una variable privada */
    #pragma omp for nowait
    for (n=2; n<=N; n++)
      facp *= n;
    #pragma omp atomic
    fact *= facp;
  }
  return fact;
}
```

The implementation `factorial3` is more efficient than the implementation `factorial2`, since the number of `atomic` operations carried out by `factorial3` is smaller.

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.

Solution:

```
int cmp(int n, double x[], double y[], int z[])
{
```

```
  int i, v, equal=0;
  double aux;
  #pragma omp parallel for private(aux,v) reduction(+:equal)
  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;
}
```

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

Solution: A parallel region should be created and the different iterations of the loop must be manually split among the different threads, according to their identifier and number of threads. Since we could not use a `reduction` clause, a possible alternative will be to protect the access to the shared variable `equal` with an `atomic` construction.

```
int cmp(int n, double x[], double y[], int z[])
{
  int i, v, equal=0, yo, nh;
  double aux;
  #pragma omp parallel private(aux,i,v,yo)
  {
    nh = omp_get_num_threads();  /* number of threads */
    yo = omp_get_thread_num();   /* thread identifier */
    for (i=yo; i<n; i+=nh) {
      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)
        #pragma omp atomic
        equal++;
    }
  }
  return equal;
}
```

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.

Solution: Usage of vector `ind` implies that several iterations of the `i` loop may end up updating the same element of vector `x`. For this reason, this update must be done in mutual exclusion, by means of the `atomic` directive.

```
#pragma omp parallel for private(s,j)
for (i=0; i<n; i++) {
   s = 0;
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   #pragma omp atomic
   x[ind[i]] += s;
}
```

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

Solution:

```
for (i=0; i<n; i++) {
   s = 0;
   #pragma omp parallel for reduction(+:s)
   for (j=0; j<i; j++) {
      s += A[i][j]*b[j];
   }
   c[i] = s;
   x[ind[i]] += s;
}
```

(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?

Solution: The `j` loop performs `i` iterations, so the iterations of loop `i` will be more costly whenever the value of `i` is larger. Therefore, a static schedule with a single chunk per thread would not be appropriate. It would be better to use a cyclic (static with `chunk`=1), dynamic or *guided* schedule.

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.

Solution: The code fragment that computes the maximum is independent from the fragment that computes the minimum, so we could consider a parallelization based on `sections`, but this option is discarded because it would exploit only 2 threads, whereas the exercise states that the parallelization must be efficient regardless of the number of threads. Therefore, we parallelize each of the loops. Given that the first loop is independent from the second one, we make use of the `nowait` clause to avoid a synchronization between them.

The solution using reductions is the following.

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

  mx = a[0];
  mn = a[0];

  #pragma omp parallel
  {
    #pragma omp for reduction(max:mx) nowait
    for (i=1;i<n;i++) {
      if (mx<a[i]) mx=a[i];
    }
    #pragma omp for reduction(min:mn)
    for (i=1;i<n;i++) {
      if (mn>a[i]) mn=a[i];
    }
    factor = mx-mn;
    #pragma omp for
    for (i=0;i<n;i++) {
      a[i]=(a[i]-mn)/factor;
    }
  }
}
```

As an alternative to reductions, it is possible to use critical sections, although it would be less efficient.

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

```
      mx = a[0];
      mn = a[0];

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

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

Solution:

```
      ...
      #pragma omp parallel
      {
        #pragma omp single
        printf("Num procs: %d\n", omp_get_num_threads());

        #pragma omp for nowait
        ...
```

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.

Solution:

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

  sum=0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:sum)
    for (i=0;i<n;i++)
      sum = sum + v[i]*v[i];
    norm = sqrt(sum);

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

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

The third parallelized loop computes a maximum and its position. The shared variables `max_pos` and `max` may be updated simultaneously in different iterations, and hence mutual exclusion is required when accessing them (`critical` construction). Furthermore, the repetition of the instruction

```
if (aux > max)
```

before the critical section aims at improving the efficiency, avoiding entering this section in cases when it is not necessary.

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

Solution: No it isn't.
It cannot be put on the first loop, since the norm cannot be computed until all threads have finished. It cannot be put on the second loop, since the third loop uses the values of `v[i]` that are updated by the second. It does not make sense to put it on the third loop, since there is no parallel work after it.

(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?

Solution: The scheduling clauses (`schedule(static,2)` or `schedule(dynamic,2)`) can be used in all `for` loops to allocate 2 by 2 iterations per thread.

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.

Solution: The easiest way of implementing it is by means of a reduction (assuming OpenMP version 3.1 or later). Also, it is necessary to synchronize the concurrent access to variable `balance`.

```
double transfers(double balance[], int origins[], int destinations[],
       double quantities[], int n)
{
   int i, i1, i2;
   double money, maxtransf=0;
   #pragma omp parallel for private(i1,i2,money) reduction(max:maxtransf)
   for (i=0; i<n; i++) {
      i1 = origins[i];
```

```
            i2 = destinations[i];
            money = quantities[i];
            #pragma omp atomic
            balance[i1] -= money;
            #pragma omp atomic
            balance[i2] += money;
            if (money>maxtransf) maxtransf = money;
         }
         return maxtransf;
      }
```

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

Solution: In this case the directive `reduction` cannot be used and it is necessary to create a critical section.

```
      double transfers(double balance[], int origins[], int destinations[],
             double quantities[], int n)
      {
         int i, i1, i2;
         double money, maxtransf=0, imax;
         #pragma omp parallel for private(i1,i2,money)
         for (i=0; i<n; i++) {
            i1 = origins[i];
            i2 = destinations[i];
            money = quantities[i];
            #pragma omp atomic
            balance[i1] -= money;
            #pragma omp atomic
            balance[i2] += money;
            if (money>maxtransf)
              #pragma omp critical
              if (money>maxtransf) {
                maxtransf = money;
                imax = i;
              }
         }
         printf("Transfer with more money=%d\n",imax);
         return maxtransf;
      }
```

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.

Solution: The program can be divided in three tasks:

- Task 1: Modification of matrix `A` from matrix `A` itself (first nested loop).
- Task 2: Modification of matrix `B` from matrix `A` (second nested loop).
- Task 3: Computation of `maxi` $= \max_{\substack{0 \le i < N \\ 0 \le j < N}} \left\{ b_{ij}^2 \right\}$, where b_{ij} is the (i, j) element of matrix `B` (third nested loop).

We can observe that Task 2 depends on Task 1 and Task 3 depends on Task 2, so each task must be finished before the next one can start. To parallelize the code, we do a loop parallelization of the three tasks:

- Task 1: Since there is a dependency in the iterations of the loop with variable `i` (the values of the new row `i` of matrix `A` depend on the previous values of the row `i-1` of matrix `A`) and it is always convenient to parallelize the outer loop, we exchange the loops and parallelize the outermost one (parallelization by columns).
- Task 2: We directly parallelize the outermost loop (computation in parallel of the rows of matrix `B`).
- Task 3: We directly parallelize the outermost loop. To obtain the value `maxi` it is possible to perform a reduction on `maxi` or use critical sections.

The next code shows the first alternative:

```
double functionp(double A[N][N],double B[N][N]) {
  int i,j;
  double aux, maxi;
  #pragma omp parallel for private(i)
  for (j=0; j<N; j++)
    for (i=1; i<N; i++)
      A[i][j] = 2.0+A[i-1][j];
  #pragma omp parallel for private(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;
      #pragma omp parallel for private(j,aux) reduction(max:maxi)
      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;
    }
```

The second alternative consists in changing the third nested loop by the following:

```
      #pragma omp parallel for private(j,aux)
      for (i=0; i<N; i++)
        for (j=0; j<N; j++) {
          aux = B[i][j]*B[i][j];
          if (aux>maxi)
            #pragma omp critical
            {
              if (aux>maxi) maxi = aux;
            }
        }
```

(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.

Solution:

$$t(N) = \sum_{i=1}^{N-1}\sum_{j=0}^{N-1} 1 + \sum_{i=0}^{N-2}\sum_{j=0}^{N-2} 1 + \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 1 \approx N^2 + N^2 + N^2 = 3N^2 \text{ flops.}$$

$$t(N,p) = \sum_{j=0}^{N/p-1}\sum_{i=1}^{N-1} 1 + \sum_{i=0}^{N/p-2}\sum_{j=0}^{N-2} 1 + \sum_{i=0}^{N/p-1}\sum_{j=0}^{N-1} 1 \approx \frac{N}{p}^2 + \frac{N}{p}^2 + \frac{N}{p}^2 = \frac{3N}{p}^2 \text{ flops.}$$

$$S(N,p) = \frac{t(N)}{t(N,p)} = \frac{3N^2}{\frac{3N}{p}^2} = p.$$

$$E(N,p) = \frac{S(N,p)}{p} = \frac{p}{p} = 1.$$

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.

Solution: In the parallelization of the second code fragment, the access to variables `k` and `positions` must be protected by means of a critical section, to avoid race conditions.

```
int funcion(double A[N][N],double positions[][2])
{
  int i,j,k=0;
  double maximum;
  /* Compute maximum */
  maximum = A[0][0];
  #pragma omp parallel
  {
    #pragma omp for private(j) reduction(max:maximum)
    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 */
    #pragma omp for private(j)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
        if (A[i][j] == maximum) {
          #pragma omp critical
          {
            positions[k][0] = i;
            positions[k][1] = j;
            k = k+1;
          }
        }
      }
    }
  }
  return k;
}
```

(b) Modify the previous parallel code so that each thread prints on the screen its identifier and the

amount of maximum values it found.

Solution: We will add variables `id` and `num_maximum`.

```
int funcion(double A[][N],double positions[][2])
{
  int i,j,k=0,id,num_maxima;
  double maximum;
  /* Compute maximum */
  maximum = A[0][0];
  #pragma omp parallel private(id,num_maxima)
  {
    #pragma omp for private(j) reduction(max:maximum)
    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 */
    id = omp_get_thread_num();
    num_maxima = 0;
    #pragma omp for private(j)
    for (i=0;i<N;i++) {
      for (j=0;j<N;j++) {
        if (A[i][j] == maximum) {
          #pragma omp critical
          {
            positions[k][0] = i;
            positions[k][1] = j;
            k = k+1;
          }
          num_maxima++;
        }
      }
    }
    printf("Thread %d: %d occurrences of the maximum value found\n",id,num_maxima);
  }
  return k;
}
```

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];
  }
  ...
}
```

Solution: In the first loop, two or more iterations may correspond to performances of the same player, in which case they would update the same entry of vectors `sum_valuation` and `num_games`. To avoid problems with race conditions, these vectors should be updated in mutual exclusion, by means of the `atomic` clause.

```
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];
  ...
  #pragma omp parallel
  {
    #pragma omp for private(player,points,rebounds,blocks)
    for (i=0;i<NP*NJ;i++) {
      player   = M[i][0];
      points   = M[i][1];
      rebounds = M[i][2];
      blocks   = M[i][3];
      #pragma omp atomic
      sum_valuation[player] += points+1.5*rebounds+2*blocks;
      #pragma omp atomic
      num_games[player]++;
      if (points>max_points) {
        #pragma omp critical
```

```
            if (points>max_points) {
              max_points = points;
              max_scorer = player;
            }
          }
        }
        #pragma omp single
        printf("Maximum scorer: player %d (%d points)\n",max_scorer,max_points);
        #pragma omp for
        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);
}
```

Solution:

```
/* 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;
  #pragma omp parallel
  {
    #pragma omp for private(i,p)
    for (k = 0; k < nv; k++) {
      i = index[k]; p = scores[k];
      #pragma omp atomic
      totals[i] += p;
      if (p > max2)
        #pragma omp critical
        if (p > max2)
          if (p > max1) { max2 = max1; max1 = p; } else max2 = p;
    }
    #pragma omp single nowait
    printf("The two highest scores have been %d and %d.\n",max1,max2);
    #pragma omp for private(t) reduction(+:pass,total)
    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.

Solution:

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

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

Solution: The use of the directive `nowait` in the first loop will not be correct, as the second loop can start only when the first loop has finished as it needs to know the number of times a song has been downloaded.

(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.

Solution:

```
void monthbilling(int n, int users[], int songs[], float prices[],
                  float billing[], int ndownloads[])
{
  int i,u,c,best_song=0;
  float p;
```

```
    int myid,downloads_thread;
    #pragma omp parallel private(myid,downloads_thread)
    {
      myid = omp_get_thread_num();
      downloads_thread = 0;
      #pragma omp for private(u,c,p)
      for (i=0;i<n;i++) {
         u = users[i];
         c = songs[i];
         p = prices[c];
         #pragma omp atomic
         billing[u] += p;
         #pragma omp atomic
         ndownloads[c]++;
         downloads_thread++;
      }
      printf("The thread %d has processed %d downloads\n", myid, downloads_thread);
      #pragma omp for
      for (i=0;i<NC;i++) {
         if (ndownloads[i]>ndownloads[best_song])
           #pragma omp critical
           if (ndownloads[i]>ndownloads[best_song])
             best_song = i;
      }
    }
    printf("Song %d has been downloaded most\n",best_song);
  }
```

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.

Solution:

```
void histogram(int histo[], float marks[], int n) {
  int i, mark;
  float rmark;
  for (i=0;i<5;i++) histo[i] = 0;
  #pragma omp parallel for private (mark, rmark)
  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 */
    #pragma omp atomic
    histo[mark]++;
  }
}
```

(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).

Solution:

```
void histogram(int histo[], float marks[], int n) {

  int i, mark, imax;
  float vmin, vmax, rmark;

  for (i=0;i<5;i++) histo[i] = 0;

  vmin = marks[0];
  vmax = marks[0];
  imax = 0;
  #pragma omp parallel for private (mark, rmark) reduction (min:vmin)
  for (i=0;i<n;i++) {
    rmark = round(marks[i]*10)/10.0;
    if (marks[i]>vmax)
      #pragma omp critical
      if (marks[i]>vmax) {
        imax = i;
        vmax = marks[i];
      }

    if (marks[i]<vmin) vmin = marks[i];

    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 */

        #pragma omp atomic
        histo[mark]++;
      }

      printf("The best mark is %f, obtained by %d, and the worst mark is %f\n",
             vmax, imax, vmin);
    }
```

Although it is less efficient, the computation of the minimum could also be done with a `critical`, similarly to the maximum (this would be necessary in old OpenMP versions that do not allow reduction with `min`). In that case, it would be convenient to use a named `critical`, for instance `critical (maximum)` and `critical (minimum)`.

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.

Solution:

```
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;
   #pragma omp parallel for private(source,dest,time) reduction(+:tavg)
   for (i=0;i<num_trips;i++) {
      source = trips[i].source_station;
      dest   = trips[i].dest_station;
      time   = trips[i].time_minutes;
      #pragma omp atomic
      num_bikes[source]--;
      #pragma omp atomic
      num_bikes[dest]++;
      tavg += time;
      if (time>tmax) {
         #pragma omp critical (maximum)
         if (time>tmax) {
            tmax=time; srcmax=source; destmax=dest;
         }
      }
      if (time<tmin) {
         #pragma omp critical (minimum)
         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);
}
```