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];
```

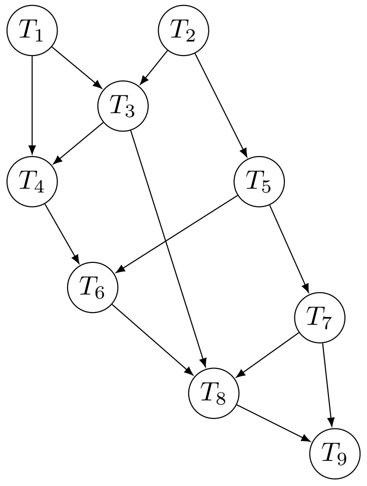
(c)
```
for (i=N-2;i>=0;i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

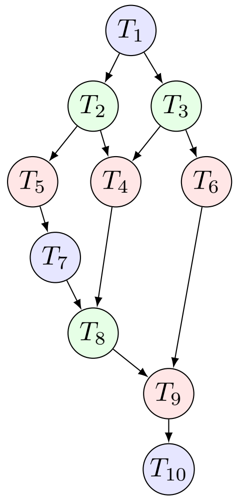
Solution: There is a data dependency among the different iterations: violates the 1st Bernstein condition ($I_j \cap O_i \neq \emptyset$), since, for instance, y[1] is an output variable of iteration i=1 and an input variable in iteration i=0. In this case it is indeed possible to remove the dependency:

```
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) shared(a,B,c)
  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: $S_p = \frac{t_1}{t_p} = \frac{2MN}{2MN/p} = p$

(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 $t_p = MN + \frac{MN}{p}$ 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

Given the following C function:

```
double fun( int n, double a[], double b[] )
{
  int i,ac,bc;
  double asum,bsum,thres;

  asum = 0; bsum = 0;
  for (i=0; i<n; i++)
    asum += a[i];
  for (i=0; i<n; i++)
    bsum += b[i];
  thres = (asum + bsum) / 2.0 / n;

  ac = 0; bc = 0;
  for (i=0; i<n; i++) {
    if (a[i]>thres) ac++;
    if (b[i]>thres) bc++;
  }
  return thres/(ac+bc);
}
```

(a) Parallelize it efficiently with OpenMP directives (without changing the existing code).

Solution:

```
double fun( int n, double a[], double b[] )
{
  int i,ac,bc;
  double asum,bsum,thres;

  asum = 0; bsum = 0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:asum) nowait
    for (i=0; i<n; i++)
      asum += a[i];
    #pragma omp for reduction(+:bsum)
    for (i=0; i<n; i++)
      bsum += b[i];
  }
  thres = (asum + bsum) / 2.0 / n;

  ac = 0; bc = 0;
  #pragma omp parallel for reduction(+:ac,bc)
  for (i=0; i<n; i++) {
    if (a[i]>thres) ac++;
    if (b[i]>thres) bc++;
  }
  return thres/(ac+bc);
}
```

(b) Indicate the a priori cost in flops, both sequential and parallel (considering only floating-point operation and assuming that a comparison implies a subtraction). What is the speed-up and efficiency for p processors? Assume that n is an exact multiple of p.

Solution:

$$
\begin{aligned}
t_1 &= \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} 1 + 3 + \sum_{i=0}^{n-1} 2 + 2 = n + n + 3 + 2n + 2 \approx 4n \quad \text{flops}\\
t_p &= \sum_{i=0}^{n/p-1} 1 + \sum_{i=0}^{n/p-1} 1 + 3 + \sum_{i=0}^{n/p-1} 2 + 2 = \frac{n}{p} + \frac{n}{p} + 3 + \frac{2n}{p} + 2 \approx \frac{4n}{p} \quad \text{flops}\\
S_p &= t_1/t_p \approx p\\
E &= S_p/p \approx 1
\end{aligned}
$$

Question 1–8

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

```
#define EPS 1e-16
#define DIMN 128
int fun(double a[DIMN][DIMN], double b[], double x[], int n, int nMax)
{
  int i, j, k;
  double err=100, aux[DIMN];

  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-16
#define DIMN 128
int fun(double a[DIMN][DIMN], double b[], double x[], int n, int nMax)
{
  int i, j, k;
  double err=100, aux[DIMN];
```

```
        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 processes 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 1–9

Given the following function:

```
#define DIM 6000
#define STEPS 6

double function1(double A[DIM][DIM], double b[DIM], double x[DIM])
{
  int i, j, k, n=DIM, steps=STEPS;
  double max=-1.0e308, q, s, x2[DIM];
  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 DIM 6000
#define STEPS 6

double function1(double A[DIM][DIM], double b[DIM], double x[DIM])
{
  int i, j, k, n=DIM, steps=STEPS;
  double max=-1.0e308, q, s, x2[DIM];

  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 `DIM` 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 1–10

Given the following function:

```
double function(double A[M][N], double b[N], double c[M], double z[N])
{
   double s, s2=0.0, elem;
   int i, j;

   for (i=0; i<M; i++) {
      s = 0.0;
      for (j=0; j<N; j++)
         s = s + A[i][j]*b[j];
      elem = s*s;
      if (elem>c[i])
         c[i] = elem;
      s2 = s2+s;
   }

   for (i=0; i<N; i++)
      z[i] = 2.0/3.0*b[i];

   return s2;
}
```

(a) Parallelize it with OpenMP in an efficient way. If possible, use a single parallel region.

Solution:

```
double function(double A[M][N], double b[N], double c[M], double z[N])
{
  ...
  #pragma omp parallel
  {
     #pragma omp for private(s,j,elem) reduction(+:s2) nowait
     for (i=0; i<M; i++) {
        ...
     #pragma omp for
     for (i=0; i<N; i++)
        ...
  }
  return s2;
}
```

(b) Modify the code so that each thread prints a line with its identifier and the number of iterations of the first `i` loop that it has performed.

Solution:

```
double function(double A[M][N], double b[N], double c[M], double z[N])
{
   ...
   #pragma omp parallel
   {
      int cont=0;
      #pragma omp for private(s,j,elem) reduction(+:s2) nowait
      for (i=0; i<M; i++) {
         cont++;
         ...
      }
      printf("Thread %d. Iterations: %d\n", omp_get_thread_num(), cont);
      #pragma omp for
      for (i=0; i<N; i++)
         ...
   }
   return s2;
}
```

(c) In the first parallelized loop, justify if one can expect differences in performance among the following schedule policies for the loop: `schedule(static)`, `schedule(static,1)`, `schedule(dynamic)`.

Solution: All iterations of the first `i` loop have the same cost, and therefore there is no reason to expect that the employed schedule policy can change the execution time (setting aside the effect that the use of cache memory may have on performance).

It would also be valid to say that the `dynamic` schedule could have a bit more overhead, since it has to make the assignment at run time.

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:

```
int search(int x[], int n, int value)
{
   int found=0, 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

Write a program that performs a loop with 100 iterations (from 0 to 99) in parallel using the schedule indicated in the environment variable `OMP_SCHEDULE`. The program must show the indices of the first and last iterations that has been performed by every thread.

For instance, if we run with 2 threads and a static schedule without chunk, the screen output should be:

```
Thread 0: first=0 last=49
Thread 1: first=50 last=99
```

Solution:

```
int main()
{
  int i,first,last;
  #pragma omp parallel private(first,last)
  {
    first = -1;
    #pragma omp for schedule(runtime)
    for (i=0;i<100;i++) {
      if (first < 0) first = i;
      last = i;
    }
    printf("Thread %d: first=%d last=%d\n",omp_get_thread_num(),primera,last);
  }
  return 0;
}
```

Question 2–4

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;
   int 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 2–5

Given the function:

```
int f(int n, double x[], double y[])
{
  int i, cont=0;
  for (i=0; i<n; i++) {
     if (x[i]>0) {
        y[i]=f2(x[i]);
        cont++;
     }
  }
  return cont;
}
```

Parallelize it, and have each thread write a message indicating its thread number and how many times it has invoked function `f2`. We want to show just one message per thread.

Solution:

```
int f(int n, double x[], double y[])
{
  int i, cont=0;

  #pragma omp parallel reduction(+:cont)
  {
     #pragma omp for
     for (i=0; i<n; i++) {
        if (x[i]>0) {
           y[i]=f2(x[i]);
           cont++;
        }
     }
     printf("Thread %d: %d invocations to f2\n", omp_get_thread_num(), cont);
  }
  return cont;
}
```

Question 2–6

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 2–7

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.

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 2–8

Given the following function:

```
double function(int n, double u[], double v[], double w[], double z[])
{
  int i;
  double sv,sw,res;

  compute_v(n,v);          /* task 1 */
  compute_w(n,w);          /* task 2 */
  compute_z(n,z);          /* task 3 */
  compute_u(n,u,v,w,z);    /* task 4 */
  sv = 0;
  for (i=0; i<n; i++)  sv = sv + v[i];         /* task 5 */
  sw = 0;
  for (i=0; i<n; i++)  sw = sw + w[i];         /* task 6 */
  res = sv+sw;
```

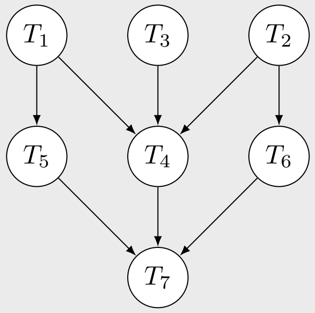
```
    for (i=0; i<n; i++)  u[i] = res*u[i];        /* task 7 */
    return res;
  }
```

The four `compute_X` functions have as input vectors those received as arguments. Those functions modify the vector included in the name of the function. Each function **only** modifies the vector included in the name. Therefore, the function `compute_u` uses vectors `v`, `w` and `z` to perform computations that are stored in the vector `u`, but none of `v`, `w`, or `z` are updated.

Therefore, functions `compute_v`, `compute_w` and `compute_z` are independent and can be performed concurrently. However, function `compute_u` must wait for the other functions to complete, since it needs (v,w,z).

(a) Draw the dependency graph for the different tasks.

Solution: The dependency graph is the following:



(b) Parallelize the function efficiently.

Solution: The parallelisation can be implemented using sections and the dependencies can be implemented with implicit barriers. All the variables are shared except the loop counter. The parallel code is:

```
double function(int n, double u[], double v[], double w[], double z[])
{
  int i;
  double sv,sw,res;

  #pragma omp parallel private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      compute_v(n,v);         /* task 1 */
      #pragma omp section
      compute_w(n,w);         /* task 2 */
      #pragma omp section
      compute_z(n,z);         /* task 3 */
    }
    #pragma omp sections
    {
      #pragma omp section
      compute_u(n,u,v,w,z);   /* task 4 */
      #pragma omp section
      {
        sv = 0;
        for (i=0; i<n; i++)  sv = sv + v[i];         /* task 5 */
      }
      #pragma omp section
      {
        sw = 0;
        for (i=0; i<n; i++)  sw = sw + w[i];         /* task 6 */
      }
    }
    #pragma omp single
    {
      res = sv+sw;
      for (i=0; i<n; i++)  u[i] = res*u[i];          /* task 7 */
    }
  }
  return res;
}
```

(c) Assuming that the cost of all the functions `compute_X` is the same and that the cost of the other loops is negligible, what will be the maximum speed-up?

Solution: Assuming that the cost of each task `compute_X` is 1, the total cost of the sequential algorithm would be 4. Considering the dependencies of the graph, we could deduce that the concurrency degree is 3 (the maximum number of tasks that can be executed in parallel is 3). The first three tasks `compute_X` can be executed in parallel, but not the last one. Therefore, the cost of the parallel algorithm is 2 if 3 or more processors are used. The maximum speed-up in this case will be 4/2=2.

Question 2–9

Parallelize the following fragment of code by means of OpenMP sections. The second argument in functions `fun_` 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 2–10

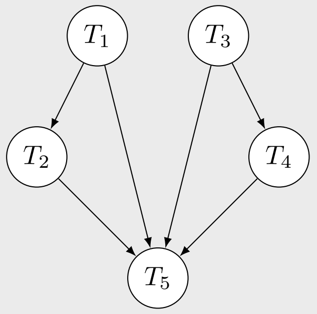
Given the following function:

```
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:



$$\begin{tikzcd}
T_1 \arrow[d] \arrow[dr] & & T_3 \arrow[dl] \arrow[d] \\
T_2 \arrow[r] & T_5 & T_4 \arrow[l]
\end{tikzcd}$$

The dependencies of `f5` with respect to the other tasks are flux dependencies (their inputs are generated by other tasks). The dependencies of `f2` and `f4` 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?

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 2–11

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 1 */
   T2(y,n);    /* Task 2 */
   T3(z,n);    /* Task 3 */

   /* Task 4 */
   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 5 */
   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 6 */
   a=s1/s2;
   res=0;
   for (i=0; i<n; i++) res+=a*z[i];
   return res;
}
```

(a) Draw the task dependency graph.

Solution:

$$\begin{tikzcd}
T_1 \arrow[dr] & & T_2 \arrow[dl] \arrow[dr] & & T_3 \arrow[dl] \\
& T_4 \arrow[dr] & & T_5 \arrow[dl] & \\
& & T_6 & &
\end{tikzcd}$$

(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 1 */
            #pragma omp section
            T2(y,n);    /* Task 2 */
            #pragma omp section
            T3(z,n);    /* Task 3 */
         }

         #pragma omp sections
         {
            #pragma omp section
            /* Task 4 */
            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 5 */
            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 6 */
         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 2–12

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.

$$\begin{tikzcd}
T_1 \arrow[dr] \arrow[drrr] & & T_2 \arrow[dl] \arrow[dr] & & T_4 \arrow[dl] \\
& T_3 \arrow[dr] & & T_5 \arrow[dl] & \\
& & T_6 & &
\end{tikzcd}$$

(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 2–13

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);
```

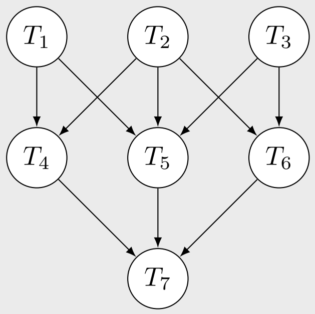
(a) Parallelize the code efficiently at the level of the loops.

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 2–14

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:

```
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 2–15

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.

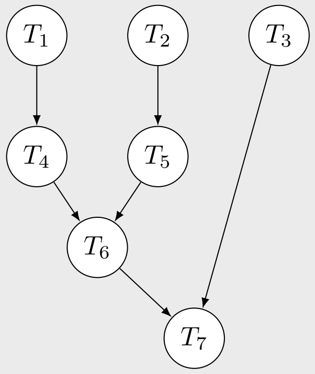
Solution:

```
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);
```

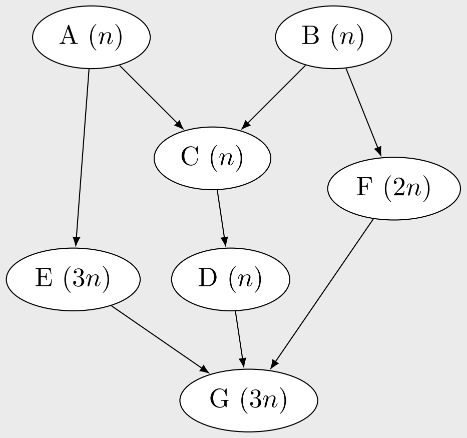
Question 2–16

Given the following function:

```
double f(int n, double vec[])
{
  double res, v[NMAX], w[NMAX];
  A(n,v,vec);          /* Copy vec in v, cost n */
  B(n,w,vec);          /* Copy vec in w, cost n */
  C(n,vec);            /* Update vec, cost n    */
  D(n,vec);            /* Update vec, cost n    */
  E(n,v);              /* Update v, cost 3n     */
  F(n,w);              /* Update w, cost 2n     */
  res = G(n,vec,v,w);  /* Compute res, cost 3n  */
  return res;
}
```

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

Solution: The dependency graph is:



The cost of each task in included in parenthesis.

The maximum concurrency degree is 3 (for instance, tasks C, E and F can be done at the same time).

The critical path is the one formed by the nodes A, E, and G. Note that even if there are paths from an initial node to a final node with a greater number of nodes, the longest (with higher cost) is the mentioned one. Its length is $L = n + 3n + 3n = 7n$ flops.

The average concurrency degree is $M = \frac{\sum_{i=A}^{G} \text{coste}_i}{L} = 12/7 = 1.714$.

(b) Parallelize it with OpenMP.

Solution: We show below a solution with `sections` where tasks C and D have been grouped.

```
double f(int n, double vec[])
{
  double res, v[NMAX], w[NMAX];
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      A(n,v,vec);
      #pragma omp section
      B(n,w,vec);
    }
    #pragma omp sections
    {
      #pragma omp section
      {C(n,vec); D(n,vec);}
      #pragma omp section
      E(n,v);
      #pragma omp section
      F(n,w);
    }
  }
  res = G(n,vec,v,w);
  return res;
}
```

(c) Compute the maximum speedup and efficiency if the code is run with 2 threads.

Solution: In order to compute the speedup and efficiency, we must first compute the sequential time (t_1) and the parallel time with 2 threads (t_2):

$$\begin{aligned} t_1 &= 12n \text{ flops} && (1)\\ t_2 &= n + 4n + 3n = 8n \text{ flops} && (2)\\ S_2 &= t_1/t_2 = 12/8 = 1.5 && (3)\\ E_2 &= S_2/2 = 1.5/2 = 75\% && (4)\end{aligned}$$

We have assumed that tasks A and B have been done in parallel and then while one thread is doing C, D and F the other one is doing E. This assignment of the second block of sections is the best case that can occur when working with 2 threads.

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 (`cont`). This variable is shared and can be concurrently accessed by all threads, producing race conditions. Therefore, it is necessary to protect it in a critical region, or better in this case with an `atomic` directive.

```
int cont = 0;
int sorted = 0;
double a;
while( !sorted ) {
  sorted = 1;
  #pragma omp parallel for private(a)
  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;
      #pragma omp atomic
      cont++;
    }
  }
  #pragma omp parallel for private(a)
  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;
      #pragma omp atomic
      cont++;
    }
  }
}
```

Another efficient option is the use of private variables in order to count the iterations of each thread and count them 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)
```

removing the `atomic` directive.

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.

Notes. 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 macro `MAX` returns the maximum of two numbers.

Solution:

```
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;
         }
      }
   }
}
```

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:

```
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. Considering that all the iterations have the same computational cost, a static distribution of the iterations can be used (default scheduling in OpenMP). 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 3.0), 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.
N.B.: 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

The cost derived from the synchronization operations is not included since it depends strongly on the values of the matrix and the execution conditions.

(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

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:

```
#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 max, min, factor;
  int i;

  max = a[0];
  for (i=1;i<n;i++) {
    if (max<a[i]) max=a[i];
  }
  min = a[0];
  for (i=1;i<n;i++) {
    if (min>a[i]) min=a[i];
  }
  factor = max-min;
  for (i=0;i<n;i++) {
    a[i]=(a[i]-min)/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 for an arbitrary number of threads.

Solution:

```
void normalize(double *a, int n)
{
  double max, min, factor;
  int i;

  max = a[0];
  min = a[0];

  #pragma omp parallel
  {
    #pragma omp for nowait
    for (i=1;i<n;i++) {
      if (max<a[i])
        #pragma omp critical (max)
        if (max<a[i]) max=a[i];
    }
    #pragma omp for
    for (i=1;i<n;i++) {
      if (min>a[i])
        #pragma omp critical (min)
```

```
            if (min>a[i]) min=a[i];
          }
          factor = max-min;
          #pragma omp for
          for (i=0;i<n;i++) {
            a[i]=(a[i]-min)/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;
      }
```

(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 use critical sections or a reduction on `maxi`.

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)
  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;
        }
    }
  return maxi;
}
```

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

```
  #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;
          }
```

(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:

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

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

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

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