Parallel Computing
Degree in Computer Science Engineering (ETSInf)
Year 2024-25 ◇ Partial exam 30/10/24 ◇ Block OpenMP ◇ Duration: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1 point)

Given the following code, where `N` and `M` are integer constants:

```
double func(double A[][N], double B[][N], double w[]) {
  int i,j,k;
  double x,wt,wk,pxc,val=1000;
  for (j=0; j<N; j++) {
    pxc=1.2;
    for (i=M; i<M+N; i++) {
      x=0.5;
      wt=0.5;
      for (k=-M; k<=M; k++) {
        wk = w[k+M];
        x += A[i+k][j]*wk;
        wt += wk;
      }
      B[i][j] = x/wt;
      pxc *= B[i][j];
    }
    if (pxc<val) val=pxc;
  }
  return val;
}
```

0.2 p. (a) Write a parallel version based on the parallelization of the innermost loop (`k`).

Solution: We would add the following directive right before the loop.

```
#pragma omp parallel for private(wk) reduction(+:x,wt)
```

0.2 p. (b) Write a parallel version based on the parallelization of the intermediate loop (`i`).

Solution: We would add the following directive right before the loop.

```
#pragma omp parallel for private(x,wt,k,wk) reduction(*:pxc)
```

0.2 p. (c) Write a parallel version based on the parallelization of the outermost loop (`j`).

Solution: We would add the following directive right before the loop.

```
#pragma omp parallel for private(pxc,i,x,wt,k,wk) reduction(min:val)
```

0.3 p. (d) Compute the sequential execution time in flops, with a step-by-step derivation.

Solution:

$$t = \sum_{j=0}^{N-1} \sum_{i=M}^{M+N-1} \left(2 + \sum_{k=-M}^{M} 3 \right) \approx \sum_{j=0}^{N-1} \sum_{i=M}^{M+N-1} 6M = \sum_{j=0}^{N-1} 6MN = 6MN^2 \text{ flops.}$$

0.1 p. (e) Suppose we have measured the execution time experimentally, getting 40 seconds with one processor and 10 seconds with 5 processors. Evaluate the corresponding speedup and efficiency.

Solution:

$$S = \frac{40}{10} = 4, \quad E = \frac{4}{5} = 0.8.$$

Question 2 (1.3 points)

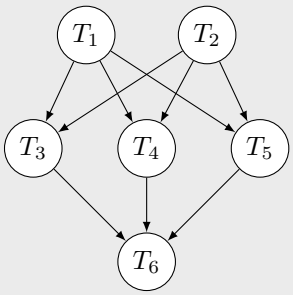
Given the following function, where `n` is a predefined constant, assume that matrices `A`, `B` and `C` have been previously filled, and also:

- The function `processcol(M,i,x)` modifies column `i` of matrix `M` from a given value `x`. Its cost is $2n$ flops.
- The system function `fabs` returns the absolute value of a floating-point number and can be considered to have a cost of 1 flop.

```
void computemat(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=0.0;
  for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
  for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];
  for (i=0;i<n;i++) processcol(A,i,alpha);
  for (i=0;i<n;i++) processcol(B,i,beta);
  for (i=0;i<n;i++) processcol(C,i,alpha*beta);
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
    }
  }
}
```

0.4 p. (a) Draw the task dependency graph, assuming there are 6 tasks corresponding to each of the `i` loops. Indicate which is the critical path and compute the average degree of concurrency of the graph.

Solution: In addition to the flow dependencies, there are anti-dependencies due to the fact that T_3, T_4 and T_5 modify the matrices, which are input to T_1 and T_2.



To obtain the average degree of concurrency, we need to compute the costs of each task:

T_1	$3n$	T_2	$4n$	T_3	$2n^2$	T_4	$2n^2$	T_5	$2n^2$	T_6	$3n^2$

Taking into account these costs, the sequential cost is:

$$t(n) = 3n + 4n + 2n^2 + 2n^2 + 2n^2 + 3n^2 = 7n + 9n^2 \approx 9n^2 \text{ flops.}$$

The critical path is $T_2 - T_4 - T_6$ (or also $T_2 - T_5 - T_6$, or $T_2 - T_3 - T_6$), whose length is:

$$L = 4n + 2n^2 + 3n^2 = 4n + 5n^2 \approx 5n^2 \text{ flops.}$$

Therefore, the average degree of concurrency is:

$$M = \frac{t(n)}{L} = \frac{9n^2}{5n^2} = \frac{9}{5} = 1,8.$$

0.5 p. (b) Implement an OpenMP parallel version with just one parallel region, based on task parallelism.

Solution: Task T_6 is not concurrent with any other task, so it can be done outside the parallel region. For the rest of the tasks we use two `sections` constructs, with 2 and 3 concurrent tasks, respectively. Variables `alpha` and `beta` are shared, since there is no possibility that different threads read and write those variables simultaneously. It is necessary to make variable `i` private explicitly because we are not using the `for` directive.

```
void computemat(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=0.0;
  #pragma omp parallel private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);           /* T1 */
      #pragma omp section
      for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];  /* T2 */
    }
    #pragma omp sections
    {
      #pragma omp section
      for (i=0;i<n;i++) processcol(A,i,alpha);       /* T3 */
      #pragma omp section
      for (i=0;i<n;i++) processcol(B,i,beta);        /* T4 */
      #pragma omp section
      for (i=0;i<n;i++) processcol(C,i,alpha*beta);  /* T5 */
    }
  }
  for (i=0;i<n;i++) {                      /* T6 */
    for (j=0;j<n;j++) {
      D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
    }
  }
}
```

0.4 p. (c) Implement an OpenMP parallel version with just one parallel region, based on loop parallelism. Whenever possible, implement optimizations such as removing unnecessary barriers.

Solution: In this case we include task 6 inside the parallel region.

```
void computemat(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=0.0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:alpha) nowait
```

```
        for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
        #pragma omp for reduction(+:beta)
        for (i=0;i<n;i++) beta += (A[i][i]+B[i][i])/2.0 - C[i][i];
        #pragma omp for nowait
        for (i=0;i<n;i++) processcol(A,i,alpha);
        #pragma omp for nowait
        for (i=0;i<n;i++) processcol(B,i,beta);
        #pragma omp for
        for (i=0;i<n;i++) processcol(C,i,alpha*beta);
        #pragma omp for private(j)
        for (i=0;i<n;i++) {
          for (j=0;j<n;j++) {
            D[i][j] = A[i][j]+0.5*B[i][j]-C[j][i];
          }
        }
      }
    }
```

To determine which barriers can be removed or nor, we can just consider the task dependencies shown in the graph. In the first loop it is also possible to eliminate the barrier, even though we have the `reduction` clause, since variable `alpha` is not used until after the barrier of the second loop.

Question 3 (1.2 points)

With the following function, the Directorate-General for Traffic manages the traffic violations done by a set of drivers. For this, the function receives as input data the vectors `Drivers`, `PointsSub` and `Quantities`, of `NI` elements, that store the identifier of the driver receiving the sanction and the license points and economic quantity associated with each of the `NI` traffic violations, respectively.

From those values, the function updates vectors `PointsDrivers` and `nSanctionsDrivers` with the points that will remain to each driver after the sanction and with the number of traffic violations they have done over time. Additionally, the function completes vector `DriversNoPoints`, which stores the identifiers of the drivers that have lost all their points. The length of that vector is returned as the function return value. Finally, the function computes and prints the largest economic sanction and the total amout of money collected in the traffic violations.

```
int process_tickets(int Drivers[],int PointsSub[],float Quantities[],
                    int PointsDrivers[],int nSanctionsDrivers[],
                    int DriversNoPoints[]) {
  int i,driver,points,quantity,nDriNoPoints=0;
  float MaxQty=0,TotalCollected=0;
  for (i=0;i<NI;i++) {
     driver=Drivers[i];
     points=PointsSub[i];
     quantity=Quantities[i];
     TotalCollected+=quantity;
     nSanctionsDrivers[driver]++;
     if (PointsDrivers[driver]>0) {
        PointsDrivers[driver]-=points;
        if (PointsDrivers[driver]<=0) {
           PointsDrivers[driver]=0;
           DriversNoPoints[nDriNoPoints]=driver;
           nDriNoPoints++;
        }
```

```
      }
      if (quantity>MaxQty)
         MaxQty=quantity;
   }
   printf("Total collected: %.2f euros\n",TotalCollected);
   printf("Maximum quantity: %.2f euros\n",MaxQty);
   return nDriNoPoints;
}
```

0.8 p. (a) Parallelize the fuction by means of OpenMP in the most efficient way.

Solution:

```
int process_tickets(int Drivers[],int PointsSub[],float Quantities[],
                    int PointsDrivers[],int nSanctionsDrivers[],
                    int DriversNoPoints[]) {
  int i,driver,points,quantity,nDriNoPoints=0;
  float MaxQty=0,TotalCollected=0;
  #pragma omp parallel for private(driver,points,quantity)
              reduction(+:TotalCollected) reduction(max:MaxQty)
  for (i=0;i<NI;i++) {
     driver=Drivers[i];
     points=PointsSub[i];
     quantity=Quantities[i];
     TotalCollected+=quantity;
     #pragma omp atomic
     nSanctionsDrivers[driver]++;
     if (PointsDrivers[driver]>0) {
        #pragma omp critical
        if (PointsDrivers[driver]>0) {
           PointsDrivers[driver]-=points;
           if (PointsDrivers[driver]<=0) {
              PointsDrivers[driver]=0;
              DriversNoPoints[nDriNoPoints]=driver;
              nDriNoPoints++;
           }
        }
     }
     if (quantity>MaxQty)
        MaxQty=quantity;
  }
  printf("Total collected: %.2f euros\n",TotalCollected);
  printf("Maximum quantity: %.2f euros\n",MaxQty);
  return nDriNoPoints;
}
```

0.4 p. (b) Parallelize the loop again, but this time without using the `for` directive (that is, doing an explicit distribution of the iterations among the threads).

Solution:

```
int process_tickets(int Drivers[],int PointsSub[],float Quantities[],
                    int PointsDrivers[],int nSanctionsDrivers[],
```

```
                    int DriversNoPoints[]) {
  int i,driver,points,quantity,nDriNoPoints=0,myid,nThreads;
  float MaxQty=0,TotalCollected=0;
  #pragma omp parallel private(driver,points,quantity,myid,i)
                  reduction(+:TotalCollected) reduction(max:MaxQty)
  {
    myid=omp_get_thread_num();
    nThreads=omp_get_num_threads();
    for (i=myid;i<NI;i+=nThreads) {
          ...
             }
          }
       }
       if (quantity>MaxQty)
          MaxQty=quantity;
    }
  }
  printf("Total collected: %.2f euros\n",TotalCollected);
  printf("Maximum quantity: %.2f euros\n",MaxQty);
  return nDriNoPoints;
}
```