Parallel Computing
Degree in Computer Science Engineering (ETSINF)
Year 2022-23 ⋄ Partial exam 9/11/22 ⋄ Block OpenMP ⋄ Duration: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1.2 points)
Given the following function:

```
double f(double A[N][N], double B[N][N], double v[N])
{
  double x,p,sigma;
  int i,j,c;
  p = 1.0;
  for (i=0; i<N; i++) {
    sigma=0;
    c=0;
    for (j=0; j<N; j++) {
      x=1.0/A[i][j];
      if (x>0) {
        c++;
        sigma+=x;
      }
    }
    for (j=0; j<=i; j++) {
      p*=B[i][j];
    }
    v[i]+=sigma/c;
  }
  return p;
}
```

0.3 p. (a) Parallelize the inner loop by means of OpenMP.

Solution: The solution is just to add the following directive right before the loop:

```
#pragma omp parallel for private(sigma,c,j,x) reduction(*:p)
```

0.5 p. (b) Parallelize the two inner loops using a single parallel region. Remove the unnecessary implicit barriers, if any.

Solution:

```
...
  c=0;              /* without changes up to this line */
  #pragma omp parallel
  {
    #pragma omp for private(x) reduction(+:c,sigma) nowait
    for (j=...) {
      ...
    }
    #pragma omp for reduction(*:p)
    for (j=...) {
```

```
        ...
      }
    }
    v[i]+=sigma/c;   /* without changes from this line on */
  ...
```

0.1 p. (c) Calculate the sequential cost, showing all the steps.

Solution:

$$t(N) = \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} 2 + \sum_{j=0}^{i} 1 + 2 \right) \approx \sum_{i=0}^{N-1} (2N + i) = \sum_{i=0}^{N-1} 2N + \sum_{i=0}^{N-1} i \approx 2N^2 + \frac{N^2}{2} = \frac{5N^2}{2} \text{flops}$$

0.3 p. (d) Suppose that we parallelize just the first `j` loop. Compute the parallel cost, showing all the steps. Calculate the speedup when p tends to infinity.

Solution: Parallel cost:

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

When p tends to infinity, $t(N,p) \approx \frac{N^2}{2}$, and therefore the speedup will be

$$S(N,p) = \frac{\frac{5N^2}{2}}{\frac{N^2}{2}} = 5$$

Question 2 (1.2 points)

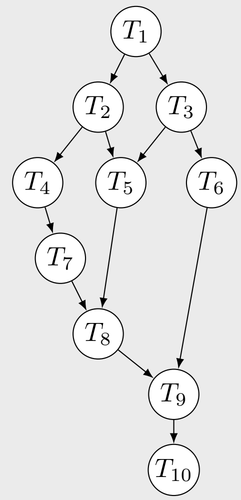
Given the following fragment of code, where **n** is a predefined constant, assuming that the matrices have been filled previously, and having into account that the three functions `f1`, `f2` and `f3` modify their second argument and have a computational cost of $\frac{1}{3}n^3$ flops, n^3 flops and $2n^3$ flops respectively, answer the following questions:

```
double A[n][n], B[n][n], C[n][n], D[n][n], E[n][n], F[n][n];

f1(n,A);       /* Task T1 */
f2(n,D,A);     /* Task T2 */
f2(n,F,A);     /* Task T3 */
f2(n,B,D);     /* Task T4 */
f3(n,E,F,D);   /* Task T5 */
f3(n,C,F,F);   /* Task T6 */
f1(n,B);       /* Task T7 */
f2(n,E,B);     /* Task T8 */
f3(n,C,E,E);   /* Task T9 */
f1(n,C);       /* Task T10 */
```

0.3 p. (a) Draw the task dependency graph.

Solution:



0.6 p. (b) Implement a parallel version by means of OpenMP using a single parallel region.

Solution: Task T_1 is not concurrent with any other task, so it can be placed outside the parallel region. Tasks T_7, T_8, T_9, and T_{10} must be necessarily executed sequentially, one after the other. Therefore, they can be left outside the parallel region. The best solution would be to aggregate tasks T_4 and T_7 so that they are done by the same thread (in the same section). In this way, task T_7 will be done in parallel with tasks T_5 and T_6.

```
f1(n,A);      /* Task T1 */
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    f2(n,D,A);   /* Task T2 */
    #pragma omp section
    f2(n,F,A);   /* Task T3 */
  }
  #pragma omp sections
  {
    #pragma omp section
    {
      f2(n,B,D);      /* Task T4 */
      f1(n,B);        /* Task T7 */
    }
    #pragma omp section
      f3(n,E,F,D);    /* Task T5 */
    #pragma omp section
      f3(n,C,F,F);    /* Task T6 */
  }
}
f2(n,E,B);    /* Task T8 */
f3(n,C,E,E);  /* Task T9 */
f1(n,C);      /* Task T10 */
```

0.3 p. (c) Obtain the speedup and efficiency of the parallel version assuming that it is executed with 4 threads in a computer with 4 processors (cores).

Solution: Sequential execution time:

$$t(n) = 3 \cdot \frac{1}{3}n^3 + 4 \cdot n^3 + 3 \cdot 2n^3 = 11n^3 \text{ flops}$$

Parallel execution time for $p = 4$:

$$t(n,p) = \frac{1}{3}n^3 + max(n^3, n^3) + max(n^3 + \frac{1}{3}n^3, 2n^3, 2n^3) + n^3 + 2n^3 + \frac{1}{3}n^3 =$$

$$\frac{1}{3}n^3 + n^3 + 2n^3 + n^3 + 2n^3 + \frac{1}{3}n^3 = \frac{20}{3}n^3; \text{flops}$$

Speedup:

$$S(n,p) = \frac{11n^3}{\frac{20}{3}n^3} = 1.65$$

Efficiency:

$$E(n,p) = \frac{1.65}{4} = 0.41$$

Question 3 (1 point)

Given the following function, where the call to function `random` returns a random integer value between the limits indicated by its arguments.

```
float value(int n)
{
  int i, j, ix, iy;
  int hit[100][100];
  float result, x, y;
  float in=0.0, out=0.0;
  int imax=0, jmax=0, max=0;

  for (i=0;i<100;i++)
    for (j=0;j<100;j++)
      hit[i][j]=0;

  for (i=0;i<n;i++) {
    ix = random(0,100);
    iy = random(0,100);
    hit[ix][iy]++;
  }

  for (i=0;i<100;i++)
    for (j=0;j<100;j++)
      if (hit[i][j]>max) {
        max = hit[i][j];
        imax=i; jmax=j;
      }
  printf("Position (%d,%d) with %d\n",imax,jmax,max);

  for (i=0;i<100;i++) {
    x = fabs(50-i)/50.0;
    for (j=0;j<100;j++) {
      y = fabs(50-j)/50.0;
      if (sqrt(x*x+y*y)<1)
        in+=hit[i][j];
      else
        out+=hit[i][j];
    }
  }
  printf("%f - %f\n", in, out);
  result = 4*in/(in+out);
  return result;
}
```

Parallelize it with OpenMP using a single parallel region, in the most efficient way.

Solution:

```
float valuepar(int n) {
  int i, j, ix, iy;
  int hit[100][100];
  float result, x, y;
  float in=0.0, out=0.0;
  int max=0, imax=0, jmax=0;

  #pragma omp parallel
  {
    #pragma omp for private (j)
    for (i=0;i<100;i++)
      for (j=0;j<100;j++)
        hit[i][j]=0;

    #pragma omp for private(ix, iy)
    for (i=0;i<n;i++) {
      ix = random(0,100);
      iy = random(0,100);
      #pragma omp atomic
      hit[ix][iy]++;
    }

    #pragma omp  for private (j)
    for (i=0;i<100;i++)
      for (j=0;j<100;j++)
        if (hit[i][j]>max)
          #pragma omp critical
          if (hit[i][j]>max) {
            max = hit[i][j];
            imax=i; jmax=j;
          }
    #pragma omp single nowait
    printf("Position (%d,%d) with %d\n",imax,jmax,max);

    #pragma omp  for private (x, j, y) reduction(+:in) reduction(+:out)
    for (i=0;i<100;i++) {
      x = fabs(50-i)/50.0;
      for (j=0;j<100;j++) {
        y = fabs(50-j)/50.0;
        if (sqrt(x*x+y*y)<1)
          in+=hit[i][j];
        else
          out+=hit[i][j];
      }
    }
  }
  printf("%f - %f = %f\n", in, out, in+out);
  result = 4*in/(in+out);

  return result;
}
```