Parallel Computing
Degree in Computer Science Engineering (ETSInf)
Year 2021-22 ◇ Partial exam 8/11/21 ◇ Block OpenMP ◇ Duration: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1.2 points)
Given the following function

```
void f(double A[N][N], double B[N][N], double C[N][N], double x[N], double z[N]) {
  int i,j;
  double sumz;

  for (i=0;i<N;i++) {
    sumz = 0;
    for (j=i;j<N;j++)
       C[i][j] = A[i][j] * x[j];
    for (j=0;j<N;j++)
       sumz += B[i][j] * x[j];
    z[i] = sumz;
  }
}
```

0.2 p. (a) Write a parallel version based on the parallelization of the outer loop.

Solution: Just above loop `i`, we must include:

```
#pragma omp parallel for private (j, sumz)
```

0.2 p. (b) Modify the previous parallel code so that it prints a single line with the number of threads that participate in the execution of the parallel loop.

Solution:

```
...
double sumz;

#pragma omp parallel
{
  #pragma omp single
  printf("Number of threads: %d\n", omp_get_num_threads());
}

#pragma omp parallel for private (j, sumz)
for (i=0;i<N;i++) {
...

// An alternative solution:

...
```

```
double sumz;

#pragma omp parallel private (j, sumz, id)
{
  #pragma omp single
  printf("Number of threads: %d\n", omp_get_num_threads());

  #pragma omp for
  for (i=0;i<N;i++) {
    ...
  }
}
```

0.5 p. (c) Write a parallel version based on the parallelization of the two inner loops, using a single parallel region. Consider using the `nowait` clause, and discuss why you use it or not.

Solution:

```
for (i=0;i<N;i++) {
  sumz = 0;
  #pragma omp parallel
  {
    #pragma omp for nowait
    for (j=i;j<N;j++)
       C[i][j] = A[i][j] * x[j];
    #pragma omp for reduction(+:sumz)
    for (j=0;j<N;j++)
       sumz += B[i][j] * x[j];
  }
  z[i] = sumz;
}
```

The `nowait` clause is used in the first loop, because it is independent from the second one.

0.3 p. (d) Compute the sequential cost and the parallel cost (show the whole derivation in both cases) assuming that we only parallelize the second one of the inner loops. Compute also the speedup and efficiency with the same assumption.

Solution:

$$t(N) = \sum_{i=0}^{N-1}(\sum_{j=i}^{N-1} 1 + \sum_{j=0}^{N-1} 2) \approx \sum_{i=0}^{N-1}(N - i + 2N) \approx 3N^2 - \frac{N^2}{2} = \frac{5N^2}{2} flops$$

$$t(N,p) = \sum_{i=0}^{N-1}(\sum_{j=i}^{N-1} 1 + \sum_{j=0}^{\frac{N}{p}-1} 2) \approx \sum_{i=0}^{N-1}(N - i + \frac{2N}{p}) \approx N^2 - \frac{N^2}{2} + \frac{2N^2}{p} = \frac{N^2}{2} + \frac{2N^2}{p} = \frac{p+4}{2p}N^2 flops$$

$$S(N,p) = \frac{\frac{5}{2}N^2}{\frac{p+4}{2p}N^2} = \frac{5p}{p+4}$$

$$E(N,p) = \frac{\frac{5p}{p+4}}{p} = \frac{5}{p+4}$$

Question 2 (1.1 points)

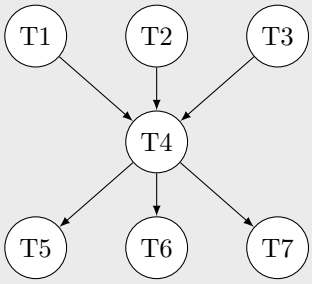
Given the following function, where `Image` is a predefined data type:

```
void transform(int n, Image im1, Image im2, float weights[3], float factor)
{
  weights[0] = channel_r(im1,im2,n);      /* Task T1, cost 5*n^2 flops */
  weights[1] = channel_g(im1,im2,n);      /* Task T2, cost 5*n^2 flops */
  weights[2] = channel_b(im1,im2,n);      /* Task T3, cost 5*n^2 flops */
  factor *= combine(weights);             /* Task T4; cost 12 flops */
  weights[0] += adjust_r(im2,n,factor);   /* Task T5, cost n^2 flops */
  weights[1] += adjust_g(im2,n,factor);   /* Task T6, cost n^2 flops */
  weights[2] += adjust_b(im2,n,factor);   /* Task T7, cost n^2 flops */
}
```

None of the functions modifies its arguments.

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

Solution:



0.5 p. (b) Implement a parallel version by means of OpenMP using only one parallel region.

Solution:

```
void transform_par(int n, Image im1, Image im2, float weights[3], float factor)
{
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      weights[0] = channel_r(im1,im2,n);
      #pragma omp section
      weights[1] = channel_g(im1,im2,n);
      #pragma omp section
      weights[2] = channel_b(im1,im2,n);
    }
    #pragma omp single
    {
      factor *= combine(weights);
    }
    #pragma omp sections
    {
      #pragma omp section
      weights[0] += adjust_r(im2,n,factor);
      #pragma omp section
```

```
            weights[1] += adjust_g(im2,n,factor);
            #pragma omp section
            weights[2] += adjust_b(im2,n,factor);
          }
        }
      }
```

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

Solution: Sequential execution time:

$$t(n) = 5n^2 + 5n^2 + 5n^2 + 12 + n^2 + n^2 + n^2 \approx 18n^2 \text{ flops}$$

Parallel execution time for $p = 4$:

$$t(n,p) = 5n^2 + 12 + n^2 \approx 6n^2 \text{ flops}$$

Speedup:

$$S(n,p) = \frac{18n^2}{6n^2} = 3$$

Efficiency:

$$E(n,p) = \frac{3}{4} = 0.75$$

Question 3 (1.2 points)

The following function updates a matrix A by adding the n values of vector *vals* (does not contain any zeros) in the positions given by the vectors *rows* and *cols* that may have repeated values.

```
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0;
  double x, max = -1e6;

  for ( k = 0 ; k < n ; k++ ) {

    i = rows[k]; j = cols[k]; x = vals[k];

    if ( x > 0 ) cp++; else cn++;

    A[i][j] += x;

    if ( x > max ) {
      max = x; row_max = i; col_max = j;
    }

  }

  printf("%d positive updates and %d negative ones.\n",cp,cn);
  printf("The largest update has been of %.1f in row %d column %d.\n",
   max, row_max, col_max );
}
```

0.8 p. (a) Parallelize the function using OpenMP.

Solution:

```
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0;
  double x, max = -1e6;

  #pragma omp parallel for private(i,j,x) reduction(+:cp,cn)
  for ( k = 0 ; k < n ; k++ ) {

    i = rows[k]; j = cols[k]; x = vals[k];

    if ( x > 0 ) cp++; else cn++;

    #pragma omp atomic
    A[i][j] += x;

    if ( x > max )
    #pragma omp critical
    if ( x > max ) {
      max = x; row_max = i; col_max = j;
    }

  }

  printf("%d positive updates and %d negative ones.\n",cp,cn);
  printf("The largest update has been of %.1f in row %d column %d.\n",
   max, row_max, col_max );
}
```

0.4 p. (b) Modify the previous parallelization so that it prints the identifier of the thread that has done more updates on matrix A, together with the number of updates.

Solution:

```
#include <omp.h>
void update( int n,int rows[],int cols[],double vals[], double A[M][N] )
{ int i,j,k, row_max,col_max, cp = 0, cn = 0, id, m = -1, c;
  double x, max = -1e6;

  #pragma omp parallel private(c)
  { c = 0;
    #pragma omp for private(i,j,x) reduction(+:cp,cn) nowait
    for ( k = 0 ; k < n ; k++ ) {

      i = rows[k]; j = cols[k]; x = vals[k];

      c++;
      if ( x > 0 ) cp++; else cn++;

      #pragma omp atomic
      A[i][j] += x;

      if ( x > max )
```

```
        #pragma omp critical
        if ( x > max ) {
          max = x; row_max = i; col_max = j;
        }

      }
      #pragma omp critical
      if ( c > m ) { m = c; id = omp_get_thread_num(); }
    }

    printf("%d positive updates and %d negative ones.\n",cp,cn);
    printf("The largest update has been of %.1f in row %d column %d.\n",
     max, row_max, col_max );

    printf("Thread %d is the one that has done more updates (%d).\n",id,m);
  }
```