Parallel Computing

Degree in Computer Science Engineering (ETSInf)

Year 2022-23 ◇ Final exam 5/2/24 ◇ Block OpenMP ◇ Duration: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1.1 points)

Given the following code:

```
double f(double A[N][N], double B[N][N], double C[N][N], double t) {
  int i,j,k,c=0;
  double f=0, s=0, m, aux;
  for (i=0; i<N; i++) {
    m=0;
    for (j=0; j<i; j++) {
      aux=0;
      for (k=0; k<N; k++)
        aux += A[i][k]*B[k][j]*B[k][j];
      C[i][j] = aux;
      if (aux>m) m = aux;
      if (aux>t) c++;
      f += aux*aux;
    }
    s += m;
  }
  return (s+f)/c;
}
```

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

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

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

0.45 p. (b) Write a parallel version based on the parallelization of loop `j`.

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

```
#pragma omp parallel for private(aux,k) reduction(max:m) reduction(+:c,f)
```

0.3 p. (c) Calculate the sequential execution time in flops, detailing the steps. Remember that comparisons between real numbers do not account for any flops.

Solution:

$$t(N) = \sum_{i=0}^{N-1}\left(\sum_{j=0}^{i-1}\left(\sum_{k=0}^{N-1} 3+2\right)+1\right)+2 \approx \sum_{i=0}^{N-1}\left(\sum_{j=0}^{i-1} 3N+1\right)+2 \approx \sum_{i=0}^{N-1} 3Ni+2 =$$

$$= 3N\sum_{i=0}^{N-1} i+2 \approx 3N\frac{N^2}{2}+2 \approx \frac{3N^3}{2}\,\text{flops}$$

Question 2 (1.2 points)

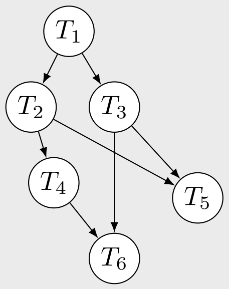
Given the following function, where `n` is a predefined constant, we assume that the matrices `A` and `B` have been filled previously, and also:

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

```
double myfun(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
  double alpha=0.0,beta=1.0;
  for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);
  for (i=0;i<n;i++) processcol(A,i,alpha);
  for (i=0;i<n;i++) processcol(B,i,alpha);
  for (i=0;i<n;i++) beta *= A[i][n-i-1];
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      C[i][j] = A[i][j]+0.5*B[i][j];
    }
  }
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      D[i][j] = beta*B[i][j];
    }
  }
}
```

0.3 p. (a) Draw the graph of data dependencies between the tasks, assuming that there are 6 tasks corresponding to each of the `i` loops.

Solution:



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

0.6 p. (b) Using the previous graph, implement a parallel version with OpenMP using a single parallel region and with task parallelism. Take into account the costs of each of the tasks to try to reduce the execution time of the parallel function.

Solution: Task T_1 is not concurrent with any other task, so it can be done outside the parallel region. As for the other tasks, the graph offers several possible implementations. To determine which one is the most suitable, the cost of the tasks must be taken into account:

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

The best implementation using `sections` constructs would be to group tasks T_4 and T_6.

```
double myfun_par(double A[n][n], double B[n][n], double C[n][n], double D[n][n]) {
  int i,j;
```

```
    double alpha=0.0,beta=1.0;
    for (i=0;i<n;i++) alpha += fabs(A[i][i]-B[i][i]);   /* T1 */
    #pragma omp parallel private(i,j)
    {
      #pragma omp sections
      {
        #pragma omp section
        for (i=0;i<n;i++) processcol(A,i,alpha);   /* T2 */
        #pragma omp section
        for (i=0;i<n;i++) processcol(B,i,alpha);   /* T3 */
      }
      #pragma omp sections
      {
        #pragma omp section
        {
          for (i=0;i<n;i++) beta *= A[i][n-i-1];   /* T4 */
          for (i=0;i<n;i++) {                      /* T6 */
            for (j=0;j<n;j++) {
              D[i][j] = beta*B[i][j];
            }
          }
        }
        #pragma omp section
        {
          for (i=0;i<n;i++) {                      /* T5 */
            for (j=0;j<n;j++) {
              C[i][j] = A[i][j]+0.5*B[i][j];
            }
          }
        }
      }
    }
  }
```

0.3 p. (c) Obtain the average degree of concurrency of the graph.

Solution: Taking into account the costs obtained previously, the sequential cost is:

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

The critical path is $T_1 - T_3 - T_5$, whose length is:

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

Therefore, the average degree of concurrency is:

$$M = \frac{t(n)}{L} = \frac{7n^2}{4n^2} = \frac{7}{4}$$

Question 3 (1.2 points)

Given the following function

```
int fun(int v[], int n) {
  int i, max, sum, ind;
  int count[100];

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

  for (i=0;i<n;i++)
     count[v[i]%100]++;

  sum = 0;
  for (i=0;i<100;i++)
     sum += count[i];
  sum /= 100;

  max = count[0];
  ind = 0;
  for (i=1;i<100;i++)
     if (count[i]>sum)
       if (count[i]>max) {
          max = count[i];
          ind = i;
       }
  return ind;

}
```

1 p. (a) Write an efficient parallel version with OpenMP. Note: it is not necessary to use a single parallel region.

Solution:

```
int funpar(int v[], int n) {
  int i, max, ind, sum=0;
  int count[100];

  #pragma omp parallel for
  for (i=0;i<100;i++)
    count[i]= 0;

  #pragma omp parallel for
  for (i=0;i<n;i++)
     #pragma omp atomic
     count[v[i]%100]++;

  sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (i=0;i<100;i++)
     sum += count[i];
  sum /= 100;

  max = count[0];
```

```
    ind = 0;
    #pragma omp parallel for
    for (i=1;i<100;i++)
       if (count[i]>sum)
         if (count[i]>max)
           #pragma omp critical
           if (count[i]>max) {
              max = count[i];
              ind = i;
           }
    return ind;

  }
```

0.2 p. (b) Write a new parallel version, as efficient as possible, in case the last part of the function (calculation of `max` and `ind`) is modified according to the following fragment of code:

```
...
max = count[0];
for (i=1;i<100;i++)
  if (count[i]>sum)
     if (count[i]>max)
        max = count[i];
return max;
```

Solution:

```
...
max = count[0];
#pragma omp parallel for reduction(max:max)
for (i=1;i<100;i++)
  if (count[i]>sum)
    if (count[i]>max)
        max = count[i];
return max;
```