Parallel Computing

Degree in Computer Science Engineering (ETSInf)

Year 2024-25 ⋄ Partial exam 27/1/25 ⋄ Block OpenMP ⋄ Duration: 1h 30m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1 point)

Given the following code:

```
double function( int n, double v[], double w[], double *a ) {
  int i, j, c = 0; double b, f = 0, e;
  for (i=0; i<n; i++) {
    e = 0;
    for (j=i+1; j<n; j++) {
      b = sqrt(v[i]*w[j]);
      if ( b > e ) e = b;
      c++;
    }
    v[i] = e;
    if ( e > f ) f = e;
  }
  *a = f;
  return c;
}
```

0.3 p. (a) Parallelize the outermost loop.

Solution: We just insert the following directive right before the first loop:

```
#pragma omp parallel for private(e,j,b) reduction(max:f) reduction(+:c)
```

0.3 p. (b) Parallelize the innermost loop.

Solution: We just insert the following directive right before the second loop:

```
#pragma omp parallel for private(b) reduction(max:e) reduction(+:c)
```

0.4 p. (c) Compute the computational cost (in flops) of the original sequential version, assuming that the `sqrt` function has a cost of 3 Flops. Taking into account the cost of a single iteration of the `i` loop, discuss whether there would be a good load balancing in case of using `schedule(static)` in section (a). Repeat the discussion in case the same scheduling was used in section (b).

Solution:

Sequential cost:

$$t(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 4 \approx \sum_{i=0}^{n-1} (4n - 4i) \approx 4n^2 - 4\frac{n^2}{2} = 2n^2 \text{ flops.}$$

The cost of an iteration of the `i` loop is approximately $4(n-i)$. Since it depends on i, the cost of each iteration is different; in particular, the first iterations are the most costly and the last ones are the least costly. Using `schedule(static)` gives place to a load imbalance, since thread 0 will be assigned the most costly block of iterations, while the last thread will be assigned the least costly block of iterations.

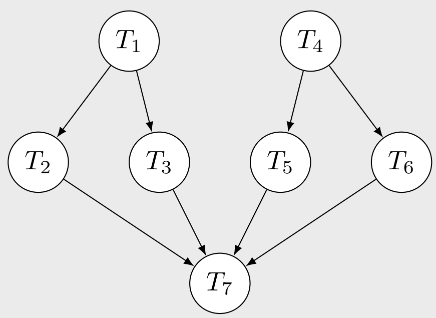
In the case of loop `j`, the cost of one iterations is 4 flops. Since all iterations cost the same, `schedule(static)` will not produce load imbalance.

Question 2 (1.4 points)
Given the following program, where function `generate` modifies its argument and has a computational cost of $6N^2$ Flops.

```
float fprocess(float A[N][N], float factor) {
  int i,j; float result=0.0;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      result+=A[i][j]/(factor+j);
  return result;
}
void processing() {
  float A[N][N], B[N][N], n1, n2, n3, n4;
  generate(A);          // T1
  n1=fprocess(A,1.1);  // T2
  n2=fprocess(A,1.2);  // T3
  generate(B);          // T4
  n3=fprocess(B,1.1);  // T5
  n4=fprocess(B,1.2);  // T6
  printf("Result: %f\n", n1+n2+n3+n4); // T7
}
```

0.4 p. (a) Obtain the task dependency graph of function `processing`, calculating the critical path and its length, as well as the average and maximum degree of concurrency.

Solution:



The critical path would be $T_1 \rightarrow T_2 \rightarrow T_7$. Function `fprocess` has a cost of $3N^2$ Flops, and therefore the critical path has a length of $9N^2 + 3$. The maximum degree of concurrency is 4 and the average degree of concurrency is $M = \frac{24N^2+3}{9N^2+3} \approx 2.67$

0.6 p. (b) Write an efficient parallel implementation of function `processing` based on sections.

Solution:

```
void processing() {
  float A[N][N], B[N][N];
  float n1, n2, n3, n4;

  #pragma omp parallel
   {
    #pragma omp sections
    {
      #pragma omp section
      generate(A);
```

```
      #pragma omp section
      generate(B);
    }
    #pragma omp sections
    {
      #pragma omp section
      n1=fprocess(A,1.1);
      #pragma omp section
      n2=fprocess(A,1.2);
      #pragma omp section
      n3=fprocess(B,1.1);
      #pragma omp section
      n4=fprocess(B,1.2);
    }
  }
  printf("Result: %f\n", n1+n2+n3+n4);
}
```

0.4 p.

(c) Compute the sequential computational cost. Compute the parallel cost, the speed-up and the efficiency in the case of using 2 threads, and also in the case of using 4 threads.

Solution:

$$t(N) = 24N^2 Flops$$

$$t(N,2) = 6N^2 + 3N^2 + 3N^2 + 3 \approx 12N^2 Flops$$

$$Sp(N,2) = \frac{24N^2}{12N^2} = 2$$

$$E(N,2) = \frac{2}{2} = 1$$

$$t(N,4) = 6N^2 + 3N^2 + 3 \approx 9N^2 Flops$$

$$Sp(N,4) = \frac{24N^2}{9N^2} = 2,67$$

$$E(N,4) = \frac{2,66}{4} = 0,67$$

Question 3 (1.1 points)

The following program obtains an approximation of the PI number from the generation of 200000 points, calculating the number of points that would lie inside or outside a circumference of radius r. From that ratio we obtain an approximation of the area and, therefore, of PI. Function `random(0,N)` returns an integer random number between 0 and $N-1$.

```
#define N 20
#define SAMPLES 200000
int main() {
  int i,j, ipos, jpos, imax, jmax, imin, jmin, r, min, max;
  int A[N][N], inside=0, outside=0;

  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        A[i][j]= 0;
```

```
  for (i=0;i<SAMPLES;i++) {
    ipos = random(0,N);
    jpos = random(0,N);
    A[ipos][jpos]++;
  }
  min = A[0][0]; max = A[0][0]; r = N/2;
  for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
       if ((r-i)*(r-i)+(r-j)*(r-j)<r*r)
         inside+=A[i][j];
       else
         outside+=A[i][j];

       if (min>A[i][j]) {
         min=A[i][j];
         imin = i;
         jmin = j;
       }
       if (max<A[i][j]) {
         max=A[i][j];
         imax = i;
         jmax = j;
       }
    }
  }
  printf("d=%d, f=%d\n", inside, outside);
  printf("Pi = %f\n", (4.0*inside)/(inside+outside));
  printf("Max A[%d][%d] = %d \n", imax,jmax,max);
  printf("Min A[%d][%d] = %d \n", imin,jmin,min);
  return 0;
}
```

0.7 p.

(a) Write a parallel implementation using OpenMP. It is not necessary to parallelize the loop that initializes matrix `A`. It is allowed to use several parallel regions.

Solution:

```
#define N 20
#define SAMPLES 200000
int main() {
  int i,j, ipos, jpos, imax, jmax, imin, jmin, r;
  int min, max;
  int A[N][N];
  int inside=0, outside=0;

  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        A[i][j]= 0;

  #pragma omp parallel for private (ipos,jpos)
  for (i=0;i<SAMPLES;i++) {
```

```
        ipos = random(0,N);
        jpos = random(0,N);
        #pragma omp atomic
        A[ipos][jpos]++;
      }
      min = A[0][0];
      max = A[0][0];
      r = N/2;
      #pragma omp parallel for private (j) reduction (+:inside, outside)
      for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
           if ((r-i)*(r-i)+(r-j)*(r-j)<r*r)
             inside+=A[i][j];
           else
             outside+=A[i][j];

           if (min>A[i][j])
             #pragma omp critical (min)
             if (min>A[i][j]) {
               min=A[i][j];
               imin = i;
               jmin = j;
             }
           if (max<A[i][j])
             #pragma omp critical (max)
             if (max<A[i][j]) {
               max=A[i][j];
               imax = i;
               jmax = j;
             }
        }
      }
      printf("d=%d, f=%d\n", inside, outside);
      printf("Pi = %f\n", (4.0*inside)/(inside+outside));
      printf("Max A[%d][%d] = %d \n", imax,jmax,max);
      printf("Min A[%d][%d] = %d \n", imin,jmin,min);
      return 0;
    }
```

0.4 p. (b) Modify the program so that it prints the value of variables `inside` and `outside` that have been computed by each of the threads.

Solution:

```
      ...
      r = N/2;
    /* Up to this point, same code as in section a */
      #pragma omp parallel private (j) reduction (+:inside, outside)
      {
         #pragma omp for
         for (i=0;i<N;i++) {
           /* Inside the loop, same code as in section a */
```

```
            ...
          }
          printf("Values of inside and outside for thread %d: %d and %d\n",
                omp_get_thread_num(), inside, outside);
        }
      /* From here, same code as the original program */
        printf("d=%d, f=%d\n", inside, outside);
        ...
```