Parallel Computing
Degree in Computer Science Engineering (ETSInf)
Year 2021-22 ◇ Partial exam 14/1/22 ◇ Block MPI ◇ Duration: 1h 45m

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Question 1 (1.3 points)
Given the following function:

```
void computev(double v[N]) {
   double A[N][N], B[N][N], C[N][N];
   readm1(A);      // T1: read a matrix A (cost N^2 flops)
   readm2(B);      // T2: read a matrix B (cost N^2 flops)
   readv(v);       // T3: read a vector v (cost N flops)
   p2Mat(A,C);     // T4
   pMatVec(B,v);   // T5
   pMatVec(C,v);   // T6
   pMatVec(B,v);   // T7
}
```

siendo

```
void p2Mat(double A[N][N],double B[N][N]){
   int i, j, k;
   for (i= 0; i< N; i++)
      for (j = 0; j < N; j++) {
          B[i][j]=0.0;
          for (k= 0; k< N; k++)
             B[i][j]+=A[i][k]*A[k][j];
      }
}

void pMatVec(double A[N][N],double v[N]){
   int i, j, k;
   double aux;
   for(k=0;k<N;k++)
      for (i= 0; i< N; i++){
         aux=v[i];
         for (j = 0; j < N; j++)
             aux+=A[i][j]*v[j]+2.0;
         v[i]=aux;
      }
}
```

0.1 p. (a) Calculate the computational cost of the functions `p2Mat` and `pMatVec`.

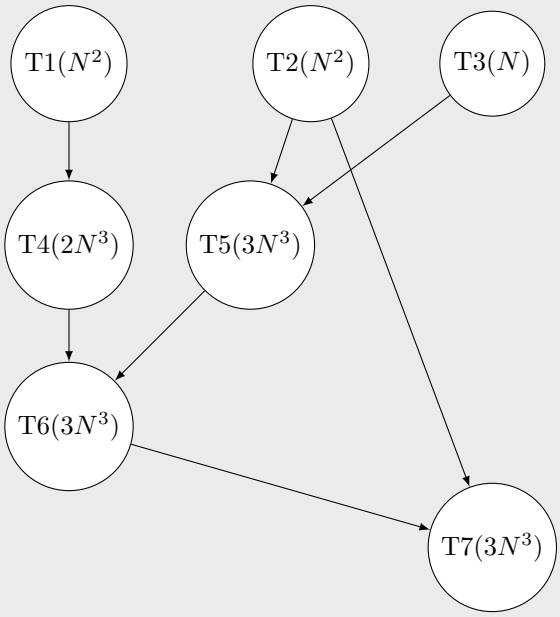
Solution:

$$\text{p2Mat}: \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1} 2 = 2N^3 \text{ flops.}$$

$$\text{pMatVec}: \sum_{k=0}^{N-1}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 3 = 3N^3 \text{ flops.}$$

0.3 p. (b) Obtain the task dependency graph.

Solution:



0.7 p. (c) Implement an MPI parallel version using only two processes, taking into account that the obtained vector v must be stored in the local memory of process P0 and assigning tasks to processes in a way that maximizes the parallelism and minimizes the cost of communications.

Solution: An assignment that maximizes the parallelism and minimizes the cost of communications is that tasks T2, T3, T5 and T7 are assigned to process P0 and tasks T1, T4 and T6 are assigned to process P1.

```
void computevp(double v[N]) {
   double A[N][N], B[N][N], C[N][N];
   int rank, p;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank==0) {
      readm2(B);      // T2: read a matrix B (cost N^2 flops)
      readv(v);       // T3: read a vector v (cost N flops)
      pMatVec(B,v);   // T5 (cost 3N^3)
      MPI_Send(v, N, MPI_DOUBLE, 1, 100,MPI_COMM_WORLD);
      MPI_Recv(v, N, MPI_DOUBLE, 1, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      pMatVec(B,v); // T7 (cost 3N^3)
   }
   else if(rank==1){
      readm1(A);      // T1: read a matrix A (cost N^2 flops)
      p2Mat(A,C);     // T4 (cost 2N^3)
      MPI_Recv(v, N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      pMatVec(C,v);   // T6  (cost 3N^3)
      MPI_Send(v, N, MPI_DOUBLE, 0, 200,MPI_COMM_WORLD);
   }
}
```

0.2 p. (d) Calculate the sequential cost and the parallel cost.

Solution:

Sequential cost:

$$t_s(N) = N^2 + N^2 + N + 2N^3 + 3N^3 + 3N^3 + 3N^3 \approx 11N^3 \text{flops}.$$

Parallel cost:

$$t_a(N,2) = N + N^2 + 3N^3 + 3N^3 + 3N^3 \approx 9N^3 \text{flops},$$
$$t_c(N,2) = 2(t_s + Nt_w),$$
$$t(N,2) = t_a(N,2) + t_c(N,2) \approx 9N^3 \text{flops} + 2(t_s + Nt_w).$$

Question 2 (1.1 points)

Given a matrix `A`, with `F` rows and `C` columns, and a vector `v` of two elements, the following function computes the vector `x`, of `F` elements.

```
void compute(double A[F][C], double v[2], double x[F]) {
  int i, j;
  double sum, min;
  /* Compute vector x
   * x[i] = sum of elements of row i of A that are >=v[0] and <=v[1] */
  for (i=0; i<F; i++) {
    sum=0;
    for (j=0; j<C; j++) {
      if (A[i][j]>=v[0] && A[i][j]<=v[1])
        sum += A[i][j];
    }
    x[i] = sum;
  }
  /* Compute the minimum element of x, and subtract it from all elements */
  min=x[0];
  for (i=1; i<F; i++)
    if (x[i]<min) min = x[i];
  for (i=0; i<F; i++)
    x[i] = x[i]-min;
}
```

0.8 p. (a) Implement a parallel version of the previous function, in which the computations are distributed in a balanced way among all processes and the communications are done by means of collective operations. Initially, both the matrix `A` and the vector `v` are available only on process 0, and we want that, at the end of the function, the vector `x` is available in all processes. You can assume that `F` is divisible between the number of processes.

The header of the function will be the following, where `Aloc` and `xloc` can be used so that each process stores its local part of `A` and `x`, respectively.

```
void computep(double A[F][C],double v[2],double x[F],
              double Aloc[][C],double xloc[])
```

Solution:

```
void computep(double A[F][C],double v[2],double x[F],
              double Aloc[][C],double xloc[]) {
  int i, j, p;
  double sum, min, minloc;
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
    MPI_Scatter(A,F/p*C,MPI_DOUBLE,Aloc,F/p*C,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Bcast(v,2,MPI_DOUBLE,0,MPI_COMM_WORLD);
    /* Compute vector x
     * x[i] = sum of elements of row i of A that are >=v[0] and <=v[1] */
    for (i=0; i<F/p; i++) {
      sum=0;
      for (j=0; j<C; j++) {
        if (Aloc[i][j]>=v[0] && Aloc[i][j]<=v[1])
          sum += Aloc[i][j];
      }
      xloc[i] = sum;
    }
    /* Compute the minimum element of x, and subtract it from all elements */
    minloc=x[0];
    for (i=1; i<F/p; i++)
      if (xloc[i]<minloc) minloc = xloc[i];
    MPI_Allreduce(&minloc,&min,1,MPI_DOUBLE,MPI_MIN,MPI_COMM_WORLD);
    for (i=0; i<F/p; i++)
      xloc[i] = xloc[i]-min;
    MPI_Allgather(xloc,F/p,MPI_DOUBLE,x,F/p,MPI_DOUBLE,MPI_COMM_WORLD);
  }
```

0.3 p.

(b) Indicate the communication cost of two different communication operations that you have used in your solution, assuming a trivial implementation of the communications.

Solution: Even though only two are requested, here is the cost of all of them:

- Scatter: $t_c = (p-1)\left(t_s + \frac{F\,C}{p} t_w\right) \approx p t_s + F\,C t_w$
- Bcast: $t_c = (p-1)(t_s + 2t_w) \approx p t_s + 2p t_w$
- Allreduce (equivalent to Reduce+Bcast): $t_c = 2(p-1)(t_s + t_w) \approx 2p t_s + 2p t_w$
- Allgather (equivalent to Gather+Bcast):
  $t_c = (p-1)\left(t_s + \frac{F}{p} t_w\right) + (p-1)\left(t_s + F t_w\right) \approx 2p t_s + pF t_w$

Question 3 (1.1 points)

The following function computes the square root of the sum of the squares of the elements of a lower triangular matrix:

```
double root_sumsquare_low_triang(double A[M][N]) {
  int i,j;
  double sum, root;
  sum=0;
  for (i=0;i<M;i++) {
     for (j=0;j<=i;j++) {
         sum+=A[i][j]*A[i][j];
     }
  }
  root=sqrt(sum);
  return root;
}
```

We want to parallelize the code in such a way that a cyclic distribution of the matrix rows is done, employing derived data types, in order to balance the workload and reduce the number of messages. We assume that:

- Process 0 initially stores the full matrix A and will have to distribute it among the rest of processes in order to perform the appropriate computations in parallel. Each process will store the rows that have been assigned to it in a local matrix A, with space for just the number of rows that have been assigned to each process, which will be declared as a parameter, together with matrix A, in the function to be implemented.
- To make things easier, matrix A will be distributed among the processes without taking into account the fact that it is lower triangular (we need not worry about sending elements equal to zero located above the diagonal, even though no process will operate with those entries).
- The return value of the function must be valid in all processes.
- Assume that the number M of rows of a matrix will always be a multiple of the number of used processes and that M and N are known constants.

Solution:

```
double root_sumsquare_low_triang(double A[M][N],double Alocal[][N]) {
  int i,j,id,np,k;
  double sum, sum_local,root;
  MPI_Datatype cyclic_rows;

  MPI_Comm_size(MPI_COMM_WORLD,&np);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  k=M/np;
  MPI_Type_vector(k,N,np*N,MPI_DOUBLE,&cyclic_rows);
  MPI_Type_commit(&cyclic_rows);
  if (id==0) {
      MPI_Sendrecv(A,1,cyclic_rows,0,0,Alocal,k*N,MPI_DOUBLE,
                          0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      for (i=1;i<np;i++)
            MPI_Send(&A[i][0],1,cyclic_rows,i,0,MPI_COMM_WORLD);
  }
  else
     MPI_Recv(Alocal,k*N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  sum_local=0;
  for (i=0;i<k;i++) {
     for (j=0;j<=np*i+id;j++) {
        sum_local+=Alocal[i][j]*Alocal[i][j];
     }
  }
  MPI_Allreduce(&sum_local,&sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  root=sqrt(sum);
  MPI_Type_free(&cyclic_rows);
  return root;
}
```