PAR – 1^{st} In-Term Exam – Course 2017/18-Q1

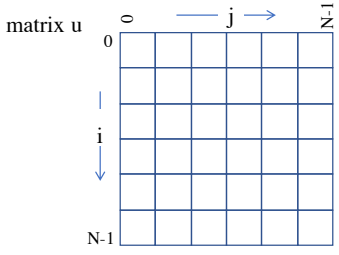
November 15^{th}, 2017

Problem 1 (4 points) Given the following code computing matrix u by blocks of $BS \times BS$ elements, following the so called *Red-Black* algorithm:

```
#define N 1024
#define BS 128
double u[N][N], residual=0.0; // residual variable only used in question 2.6

void compute_block(int ii, int jj) {
    double tmp;

    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=max(1, jj); j<min(jj+BS, N-1); j++) {
            tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            u[i][j] = tmp/4;
        }
}

void main() {
    ...
    // RED loop: traversing all RED blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=(ii%2)*BS; jj<N; jj+=2*BS)
            // Computing a RED block of BSxBS elements
            compute_block(ii, jj);

    // BLACK loop: traversing all BLACK blocks
    for (int ii=0; ii<N; ii+=BS)
        for (int jj=((ii+1)%2)*BS; jj<N; jj+=2*BS)
            // Computing a BLACK block of BSxBS elements
            compute_block(ii, jj);

    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

Observe that the so called *RED* loop computes half of the blocks while the so called *BLACK* loop computes the other half of the blocks. Based on this observation and the assumption that `2*BS` perfectly divides `N`, **we ask you:**

1. In the following diagram representing the blocks of matrix u computed in the previous code (assuming `N=6*BS`), indicate which blocks are computed by the *RED* and *BLACK* loops.



Label:
- R: block computed by *RED* loop
- B: block computed by *BLACK* loop

2. If each invocation of function `compute_block` is defined as a task, are there any dependences precluding the parallel execution of the tasks generated during the execution of the *RED* loop? And for the tasks generated in the *BLACK* loop? Are there any dependences between the tasks generated in the *RED* loop and the tasks generated in the *BLACK* loop? If your answer to any of the previous questions is affirmative, please clearly indicate what is causing those dependences.

3. Assuming the definition of task above, write a first parallel version of the code making use of the following OpenMP constructs: `parallel`, `single`, `task` without `depend` clauses, `taskwait` and/or `taskgroup`.

4. Write a second parallel version of the code making use of the following OpenMP constructs: `parallel`, `single` and `task` with `depend` clauses.

5. If the previous *RED* and *BLACK* loops are surrounded by an iterative loop, as follows:

```
void main() {
    ...
    residual = 0.0;
```

```
    for (int iter=0; iter < MAXITER; iter++) {
        // RED loop: traversing all RED blocks
        ...
        // BLACK loop: traversing all BLACK blocks
        ...
    }

    printf("Value for residual = %f and central element %f \n", residual, u[N/2][n/2]);
    ...
}
```

Do you have to do any modification to your second parallel version (tasks with `depend` clauses) in order to make the parallel execution correct. If your answer is negative justify why; if affirmative, do the necessary changes.

6. If function `compute_block` is redefined as follows:

```
void compute_block(int ii, int jj) {
    double tmp, diff;

    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=max(1, jj); j<min(jj+BS, N-1); j++) {
            tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            diff = tmp - u[i][j];
            residual += diff * diff;
            u[i][j] = tmp/4;
        }
}
```

Add the required synchronization or data sharing constructs to guarantee the proper update of variable `residual`, **with the following constraint:** the overhead due to synchronization or data sharing should be kept to the MINIMUM, if possible to be executed ONLY once per task instantiation.

Problem 2 (1 point) For the loop below executed inside the scope of an OpenMP parallel region:

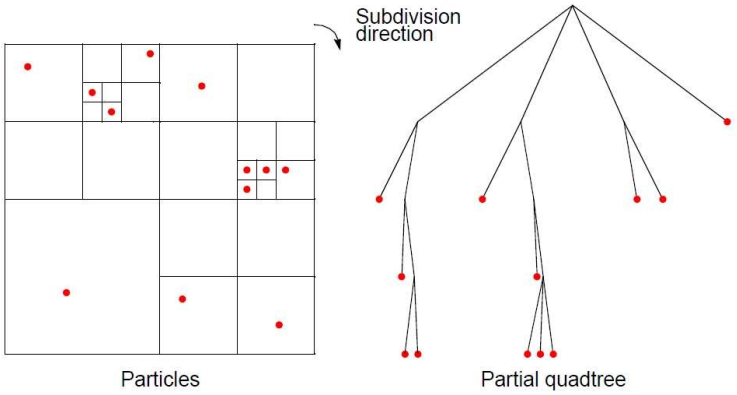
```
#pragma omp for schedule(dynamic, p)
for (i = 0; i < n; i++)
    b[i] = foo(a[i]);
```

Write two equivalent versions (equivalent in terms of number of chunks and number of iterations per chunk) in which:

- **version 1**: You make use of the `taskloop` construct.
- **version 2**: You make use of the `task` construct.

Note: assume that `p` exactly divides `n`.

Problem 3 (5 points) Let us consider the use of a *quadtree* data structure to store information about particles in a 2-dimensional space. The space is divided into four quadrants, and a subtree is used to store the information corresponding to the particles in each quadrant. The *quadtree* is recursively defined by dividing each quadrant in four sub-quadrants recursively until the leaves are reached (which store the information of a single particle). To save space and computations, a branch of a subtree at any level is cut when there are no particles in the corresponding sub-quadrant. Thus, we deal with *partial quadtrees* as the one shown in the example below which has to be used to answer the questions in this problem:



We are given the following OpenMP parallel code which traverses a partial *quadtree* and updates the data items (particles) at the leaves of the *quadtree*. For each leaf, the update is the result of computing the force between the particle represented by that leaf and the reference particle provided as the second parameter in subroutine `QT_Traversal`:

```
TreeNode *root, *ref_particle;

void QT_Traversal(TreeNode *subTree, TreeNode *particle) {
                                      // particle is not included in subTree
  TreeNode *TreeQuadrant;
  for(i=0; i<4; i++) {
      TreeQuadrant = subTree.quadrant[i];
      if (TreeQuadrant != NULL)       // if this branch is not cut
          if (TreeQuadrant.isLeaf)    // subtree in this branch is a leaf
              #pragma omp task
              ComputeForces(TreeQuadrant, particle);
          else
              #pragma omp task
              QT_Traversal(TreeQuadrant, particle);
  }
}
void main() {
  Initialization (root, ref_particle); // Inherently SEQUENTIAL code
  #pragma omp parallel
  #pragma omp single
  #pragma omp task
  QT_Traversal(root, ref_particle);    // Inherently PARALLELIZABLE code
  QT_Write(root);                      // Inherently SEQUENTIAL code
}
```

The `Initialization` and `QT_Write` functions constitute the non-parallelizable part of the program. In the parallelizable part of the program, `QT_Traversal` tasks recursively traverse the *quadtree* and `ComputeForces` tasks are in charge of doing the actual computation.

1. Assuming that the execution of each `QT_Traversal` and each `ComputeForces` invocation takes 3 and 9 time units, respectively, **we ask you** to compute the values of T_1, T_∞ and $Parallelism$ for the Task Dependence Graph (TDG) shown in the last page of the exam (which would be generated by the program when traversing the *quadtree* shown above). **Notes:** 1) to answer this question you should NOT consider the assignment of tasks to processors shown in that TDG; and 2) you should NOT consider any overhead due to task creation/synchronization.

2. If the execution of the non-parallelizable part of the program takes 33 time units, **we ask you** to compute the value of the parallel fraction ϕ for the whole program and the ideal speed–up S_∞ that would be achieved for the whole program if we were able to scale the parallelizable part to infinite processors.

3. Assume that tasks are assigned to 4 processors ($P_0 \ldots P_3$) as shown in the last page of the exam. The task executed at the root of the *quadtree* (i.e. the initial call to `QT_Traversal` in the main program) is assigned to P_0. Then the 4 `QT_Traversal` tasks that are generated at the first level of recursion are assigned in such a way that P_i gets assigned the task generated at iteration i (quadrants traversed in clockwise order). After that, the whole subtree of tasks generated from this first-level task is assigned to the same processor P_i. Considering the same task durations and execution time of the non–parallelizable part as in the previous questions, **we ask you** to compute the values for T_4 and S_4 for the whole program.

4. Assume that the information for all particles in the leaf nodes of the *quadtree* are stored in the memory of processor P_0. This means that all other processors will have to access to this information remotely in order to execute `ComputeForces` tasks. More specifically, function `ComputeForces` has to first read the information associated to the particle in the leaf, which occupies 20 bytes in total, do the computation, and finally write the updated leaf particle, as shown below:

```
void ComputeForces(TreeNode *subTree, TreeNode *particle) {
  TreeNode p = ReadParticle(subTree);
  Compute(&p, particle);
  WriteParticle(&p, subTree);
}
```

Observe that remote accesses are performed during the execution of the task, neither before nor after. Assuming that 1) the access to data stored in a remote processor adds an overhead of $t_{overhead} =$

$t_s + m \times t_w$ (being $t_s = 1$ and $t_w = 0.1$ the start-up time and per-byte transfer time, respectively); 2) a processor can only perform one remote access at a time, serve one remote access at a time, but can do both of them at the same time; and 3) if 2 or more processors want to simultaneously perform a remote access to the same destination processor, the request of the processor P_i with lower identifier (i=1..3) will be served first. **We ask:**

(a) Compute the minimum remote data access overhead for any of the `ComputeForces` tasks assigned to P_i (i=1..3).

(b) Assuming the restrictions in the number of remote accesses simultaneously supported by a processor; that the execution of each `QT_Traversal` takes 3 time units, and each `ComputeForces` invocation takes 9 time units plus the time necessary for remote data transfers (according to the remote data access overheads you just computed) while the rest of the work has negligible execution time; and considering the assignment of tasks to processors described in question 3 and the task identifiers given in the picture shown in the last page of the exam, **we ask you** to draw the temporal diagram with a scheduling of tasks in each processor that results in the BEST possible execution time of the previous OpenMP parallel program. Answer this question by filling in the temporal diagram provided in the last page of this exam.

5. Starting from the OpenMP parallel program provided above, **we ask** you to do all the necessary changes to include a cut–off mechanism that ensures that no additional `QT_Traversal` tasks are created after a certain recursion level (`MAX_LEVEL`) (**important:** the cut-off mechanism should only apply to `QT_Traversal` tasks, not to `ComputeForces` tasks that should always be generated).

TDG (with task numbering) and empty temporal diagram for **Problem 3**. Use the temporal diagram to answer to **question 4.b**, filling each cell with: 1) the number of the task that is executed; 2) R if a remote read data transfer is occurring; or 3) W if a remote write data transfer is occurring. **Important:** each slot in the temporal diagram corresponds to 3 time units.