PAR – 1^{st} In-Term Exam – Course 2017/18-Q2

April 18^{th}, 2018

Problem 1 (3 points) Given the following C code with tasks identified using the *Tareador* API:

```
#define N 4
int m[N][N];

// initialization
for (int i=0; i<N; i++) {
    tareador_start_task ("for_initialize");
    for (int k=i; k<N; k++) {
        if (k == i) modify_d(&m[i][i], i, i);
        else {
            modify_nd (&m[i][k], i, k);
            modify_nd (&m[k][i], k, i);
        }
    }
    tareador_end_task ("for-initialize");
}

// computation
for (int i=0; i<N; i++) {
    tareador_start_task ("for_compute");
    for (int k=i+1; k<N; k++) {
        int tmp = m[i][k];
        m[i][k] = m[k][i];
        m[k][i] = tmp;
    }
    tareador_end_task ("for-compute");
}

// print results
tareador_star_task ("output");
print_results(m);
tareador_end_task ("output");
```

Assuming that: 1) the execution of the `modify_d` routine takes 10 time units and the execution of the `modify_nd` routines takes 5 time units; 2) each internal iteration of the computation loop (i.e. each internal iteration of the *for_ compute* task) takes 5 time units; and 3) the execution of the *output* task takes 100 time units, **we ask:**

1. Draw the task dependence graph (TDG), indicating for each node its cost in terms of execution time (in time units).

2. Compute the values for T_1, T_∞, the parallel fraction (phi) as well as the potential parallelism.

3. Indicate which would be the most appropriate task assignment on two processors in order to obtain the best possible "speed up". Calculate T_2 and S_2.

Problem 2 (4 points) Given the following C code:

```
#define VEC_SIZE        1000000000
#define MIN_SIZE        10
#define MAX_TASKS       200
#define MAX_DEPTH       50

int compute_basic(int size, int *V) {
        int ret = 0;
        for (int i=0; i<size; i++) {
                ret+=foo(V[i]);
        }
        return ret;
}

int compute_rec(int size, int *V) {
        int ret = 0;
        int ret1, ret2;
        if (size > MIN_SIZE) {
                ret1 = compute_rec(size/2,V);
                ret2 = compute_rec(size/2,&V[size-size/2]);
                ret = ret1 + ret2;
        } else ret = compute_basic(size, V);
        return ret;
}

void init_vec(int size, int *V) {
        for (int i=0; i<5; i++) V[i] = i;
        // main loop
        for (int i=5; i<size; i++) V[i] = foo2(V[i-5]);
}

void main(int argc, char *argv[]) {
        int ret;
        int my_vec[VEC_SIZE];
        init_vec(VEC_SIZE, my_vec);
        ret = compute_rec(VEC_SIZE, my_vec);
        printf("%d\n", ret);
}
```

Comment: the execution of functions `foo` and `foo2` return some values based on their input arguments (which are not modified).

1. Create a parallel version in OpenMP using a recursive task decomposition for the `compute_rec` function. (In this first version you don't have to include any cut-off mechanism). Select the most appropriate strategy (tree or leaf) that will maximize the processor utilisation assuming a system with a high number of processors.

2. Implement a task generation control mechanism based on the depth level, making use of the appropriate clauses for the OpenMP `task` construct. Use `MAX_DEPTH` as the maximum depth level to decide if tasks must be created or not.

3. Implement a task generation control mechanism based on the number of pending tasks to be executed. Use `MAX_TASKS` as the maximum number of tasks pending to be executed to decide if we have to create a new task or not.

4. After evaluating the performance obtained, we detect the function `init_vec` consumes a lot of time and we decide to paralelize it. Propose a paralelization for the main loop in `init_vec` using work-sharing constructs to extract the maximum parallelism. Justify the selected loop scheduling.

Problem 3 (3 points) Given the following C code:

```
#define N 1024
#define BS 256
#define N_ITER 2

double u1[N][N];
double u2[N][N];

void compute_block(double A[N][N], double B[N][N], int jj) {
   double tmp;
      for (int j=max(1, jj); j<min(jj+BS, N-1); j++)
       for (int i=1; i<N-1; i++) {
         tmp = A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1] - 4*A[i][j];
         B[i][j] = tmp/4;
      }
}

void my_print(double A[N][N]) {
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
       printf("(%d,%d)=%e\n",i,j,A[i][j]);
}

void main() {
...
 for (i=0; i<N_ITER; i++) {
  // loop 1
  for (int jj=0; jj<N; jj+=BS)
    compute_block(u1, u2, jj);
  // loop 2
  for (int jj=0; jj<N; jj+=BS)
    compute_block(u2, u1, jj);
 }

 // Only processor 0 should execute this function
 my_print(u1);
...
}
```

Consider that each call to `compute_block` is a task, `my_print` is also a task only executed by processor 0 and matrices are initially distributed as indicated below. **We ask:** Write the expression that determines the execution time T_4, clearly indicating the contribution of the computation time $T_{4(comp)}$ and data sharing overhead $T_{4(mov)}$, for the following assignment of tasks to processors in each of the two iterations of `loop i` of the main program:

Task	Processor
loop1 jj=0, loop2 jj=0 and my_print(u1)	0
loop1 jj=256 and loop2 jj=256	1
loop1 jj=512 and loop2 jj=512	2
loop1 jj=768 and loop2 jj=768	3

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrices `u1` and `u2` are initially distributed by columns (BS consecutive columns per processor, where $BS = N/P$ and $P = 4$); 3) data sharing model with $t_{comm} = t_s + m \times t_w$, being t_s y t_w the start–up time and transfer time of one element, respectively; and 4) the execution time for a single iteration of the innermost loop body takes t_c time units (invocation to `compute_block`) and each call to the `printf` library function takes $2 \times t_c$ time units.