PAR – In-Term Exam – Course 2023/24-Q1

November 2^{nd}, 2023

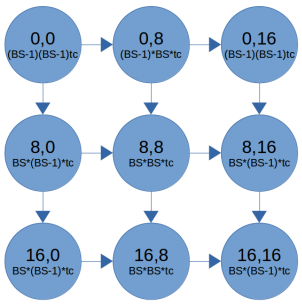
Problem 1 (5.0 points) Given the following code:

```
#define BS 8
#define N  24
double A[N][N];
int ii, jj, i, j;
...

for (ii=0; ii<N; ii+=BS)
  for (jj=0; jj<N; jj+=BS) {
     tareador_start_task("Comp_ii_jj");
     for (i=max(1,ii); i<min(ii+BS,N); i++)
       for (j=max(1,jj); j<min(jj+BS,N-1); j++)
          A[i][j] = A[i][j-1] + A[i-1][j] + A[i][j+1]; // cost of this line: tc t.u.
     tareador_end_task("Comp_ii_jj");
  }
...
```

1. (1 point) Draw the Task Dependence Graph (TDG) based on the above Tareador task definitions and for `BS=8` and `N=24`. Each task should be clearly labeled with the values of ii, jj and its cost in time units.

   Solution:



2. (2.0 points) Compute the values for T_1, T_∞ and P_{min}. Draw the temporal diagram for the execution of the TDG in the previous question on P_{min} processors. As indicated, consider the cost of the innermost loop body to be t_c time units.

   Solution:

   $T_1 = (N-1)(N-2) \times t_c$

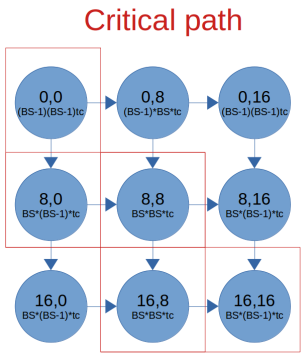
   $T_\infty = ((BS-1)^2 + (BS)(BS-1) + BS^2 + BS^2 + (BS)(BS-1)) \times t_c$

   $P_{min} = 3$

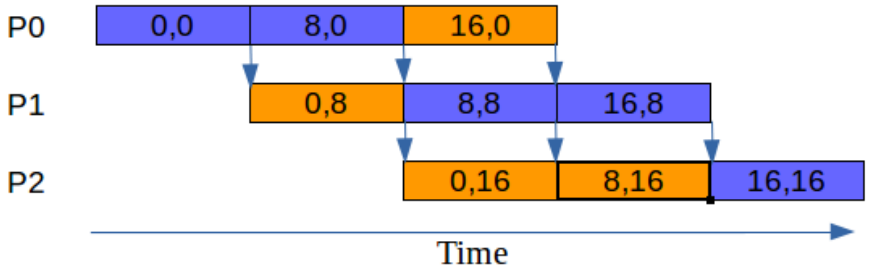
   Considering the particular values of `N` and `BS` provided in the statement of the exercise this translates into:

   $T_1 = 506 \times t_c$

   $T_\infty = 289 \times t_c$



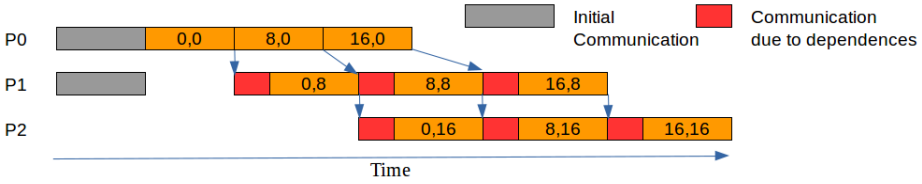
We consider $(BS-1) \approx BS$ in order to simplify the time diagram:



3. (2.0 points) Assume the same task definition, and consider a distributed memory architecture with P processors, $BS = N/P$, and N a very large value multiple of P (you can assume $BS-1$ is approximately the same as BS to simplify the model). Let's assume that matrix A is initially distributed by columns (N/P consecutive columns per processor) and tasks are scheduled so that a task is executed in the processor that stores the data the task has to update.

   We ask you to:

   (a) (1.0 points) Draw the time diagram for the execution of the tasks in P processors clearly identifying the computation and the data sharing time. Note: you can assume $P = 3$ for this question a).



   Initial communication corresponds to the access of the processor p to the 1st column, of N elements, of the processor $p+1$ due to access $A[i][j+1]$ (last processor has no initial communication). And for communications due to dependencies ($A[i][j-1]$), each processor p has to read a chunk of BS elements of the last column calculated by processor $p-1$ before starting each of its tasks (except for the first processor).

   (b) (1.0 points) Write the expression that determines the execution time, T_p as a function of N and P clearly identifying the contribution of the computation time and the data sharing overheads, assuming the data sharing model explained in class in which the overhead to perform a remote memory access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred; at a given time, a processor can only perform one remote access to another processor and serve one remote access from another processor.

   Solution: Note: $(BS-1) \rightarrow BS$, and $BS = N/P$.

$$T_P = T_{comp} + T_{comm}$$
$$T_{task} = BS \times \frac{N}{P} \times t_c = \frac{N}{P} \times \frac{N}{P} \times t_c$$
$$T_{comp} = T_{task} \times (\frac{N}{BS} + P - 1) = T_{task} \times (P + P - 1)$$
$$T_{comm} = t_s + N \times t_w + (P + P - 2)(t_s + BS \times t_w)$$

Problem 2 (5.0 points) Given the following sequential recursive algorithm:

```
#define N 1024
#define NSTATES 128
#define MINROWS 2

int histogram[NSTATES];

int base_processing (int data[N][N], int start, int nrows) {
     int outofrange=0;

     for (int i=start; i<start+nrows; i++) {
       for (int k=0; k<N; k++) {
          int value = compute (data[i][k]); /* perform computation on the parameter */
          if (value >= NSTATES)
               outofrange++;
          else if (value >= 0)
               histogram[value]++;
       }
     }
     return outofrange;
}

int rec_processing (int data[N][N], int start, int nrows) {
    int res1, res2=0;

    if (nrows < MINROWS)
            res1 = base_processing (data, start, nrows);
    else {
            res1 = rec_processing (data, start, nrows/2);
            res2 = rec_processing (data, start+nrows/2, nrows-nrows/2);
    }
    return res1 + res2;
}

int main() {
  int data[N][N];
  ...
  int res = rec_processing(data, 0, N);
  ...
}
```

We ask you to answer the following **independent** questions:

1. (2.5 points) Write an OpenMP parallel version of the `base_processing` function, following an *Iterative* Task Decomposition, making use of the OpenMP explicit tasks. Your implementation should minimize synchronization overheads and take into account **the potential imbalance** generated by the `compute` function.

Some considerations about the all the proposed solutions:

- All of them have `parallel` and `single`.
- All of them create explicit tasks.
- All of them avoid creating tasks with only one iteration of k (too much task creation overhead).
- All of them avoid creating tasks doing full loop k (too much imbalance due to compute).
- All of them avoid waiting for all tasks at each iteration of i (we are looking for parallelism). I.e. nogroup should be used in the taskloop, but them reduction is not allowed.
- All of them perform a reduction of `outofrange` variable (avoid data race condition and reduce data synchronization).
- All of them perform an of `atomic` to update histogram (reduction could be possible also but it may be costy in memory).

Possible Solution 1: Using explicit tasks with hand-made reduction

```
#define GRANULARITY 4

int base_processing (int data[N][N], int start, int nrows) {
    int outofrange=0;
    #pragma omp parallel
    #pragma omp single
    {
      for (int i=start; i<start+nrows; i++) {
        for (int kk=0; kk<N; kk+=GRANULARITY) {
          #pragma omp task firstprivate(kk)
          {
           int tmp_outofrange = 0;
           for (int k=kk; k<min(kk+GRANULARITY,N); k++) {
            int value = compute (data[i][k]); /* perform computation on the parameter */
            if (value >= NSTATES)
                tmp_outofrange++;
            else if (value >= 0)
                #pragma omp atomic
                histogram[value]++;
           }
           #pragma omp atomic
           outofrange+=tmp_outofrange;
          }
        }
    }
   }
   return outofrange;
}
```

Some considerations about the implementation proposal:

- We do not create one task per each k-loop iteration to avoid too much task creation overhead. Instead, we have done strip-mining of the k-loop to create one task per each `GRANULARITY` number of iterations of k loop. This number of iterations has been set to 4.
- There is a possible data race condition updating variable `outofrange`. We do a hand-made reduction of variable `outofrange`: `tmp_outofrange` local variable is used to avoid data race conditions and, after the `GRANULARITY` iterations, we update global variable `outofrange`. This way only one atomic per task is paid (`GRANULARITY` times better than one atomic per k iteration).

- There is a possible data race condition updating variable `histogram[value]`. We use `atomic` to update it. The atomic protection on the histogram update may generate synchronization overhead depending on how often each position is updated. A workaround to reduce this synchronization would be to have local histograms for each task and combine it later, outside the loop. This could be much costly depending on the size of the histogram.

Possible Solution 2: Using explicit tasks with taskgroup and task_reduction

This solution is equivalent to previous one. In this solution `task_reduction` and `in_reduction` clauses are used instead of a hand-made reduction.

```
#define GRANULARITY 4

int base_processing (int data[N][N], int start, int nrows) {
   int outofrange=0;
   #pragma omp parallel
   #pragma omp single
   {
     #pragma omp taskgroup task_reduction(+:outofrange)
     {
      for (int i=start; i<start+nrows; i++) {
        for (int kk=0; kk<N; kk+=GRANULARITY) {
          #pragma omp task firstprivate(kk) in_reduction(+:outofrange)
          {
           for (int k=kk; k<min((kk+GRANULARITY),N); k++) {
            int value = compute (data[i][k]); /* perform computation on the parameter */
            if (value >= NSTATES)
                outofrange++;
            else if (value >= 0)
                #pragma omp atomic
                histogram[value]++;
           }
          }
        }
      }
     }
   }
   return outofrange;
}
```

Some considerations about the implementation proposal:

- `taskgroup` is not done at the *kk*-loop level because we do not want to wait for the sibling tasks at the end of each i iteration.

Possible Solution 3: Using taskloop in_reduction nogroup + taskgroup task_reduction

This solution includes a in_reduction, and it is equivalent to Solution 2.

```
#define GRANULARITY 4

int base_processing (int data[N][N], int start, int nrows)
{
    int outofrange=0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup task_reduction(+:outofrange)
  {
    for (int i=start; i<start+nrows; i++) {
          #pragma omp taskloop grainsize(GRANULARITY) firstprivate(i)\
                               in_reduction(+:outofrange) nogroup
          {
          for (int k=0; k<N; k++) {
            int value = compute (data[i][k]); /* perform computation on the parameter */
            if (value >= NSTATES)
                outofrange++;
            else if (value >= 0)
               #pragma omp atomic
                histogram[value]++;
          }
         }
    }
   }
}
    return outofrange;
}
```

Some considerations about the implementation proposal:

- `taskloop nogroup` is used to avoid waiting for all tasks at each i iteration.
- `taskloop in_reduction` is used instead of `reduction` due to the `nogroup` clause. This clause doesn't allow to use reduction alone. Instead, `in_reduction` makes each created task with `taskloop` contribute to the `task_reduction` of the `taskgroup`.

Alternative Non-expected Solution: Using taskloop with collapse - not seen at class

This solution includes a reduction, and it is equivalent to Solutions 2 and 3.

```
#define GRANULARITY 4
int base_processing (int data[N][N], int start, int nrows) {
    int outofrange=0;
#pragma omp parallel
#pragma omp single
#pragma omp taskloop collapse(2) reduction(+:outofrange) grainsize(GRANULARITY)
    for (int i=start; i<start+nrows; i++) {
        for (int k=0; k<N; k++) {
            int value = compute (data[i][k]); /* perform computation on the parameter */
            if (value >= NSTATES)
                outofrange++;
            else if (value >= 0)
                #pragma omp atomic
                histogram[value]++;
        }
    }
    return outofrange;
}
```

Some considerations about the implementation proposal:

- `taskloop collapse(2)` creates tasks of `GRANULARITY` iterations of the collapsed `i + k` loops. `collapse(2)` joins 2 loops (i and k) in only one loop with $nrows \times N$ iterations.

2. (2.5 points) Write an OpenMP parallel version of the sequential recursive program, following a *Recursive* Task Decomposition using the *Tree* strategy. The implementation should take into account the overhead due to task creation, by limiting their creation once a certain level in the recursive tree is reached.

Solution:

```
#define MAXDEPTH X  // Any value, not specified in the exercise

int base_processing (int data[N][N], int start, int nrows)
{
     int outofrange=0;

     for (int i=start; i<start+nrows; i++) {
        for (int k=0; k<N; k++) {
          int value = data[i][k];
          if (value >= NSTATES)
               outofrange++;
          else if (value >= 0)
               #pragma omp atomic
               histogram[value]++;
       }
     }
     return outofrange;
}

int rec_processing (int data[N][N], int start, int nrows, int depth)
{
    int res1, res2=0;

    if (nrows < MINROWS)
            res1 = base_processing (data, start, nrows);
    else {
         if (!omp_in_final()) {
            #pragma omp task final (depth >= MAXDEPTH) shared (res1)
            res1 = rec_processing (data, start, nrows/2, depth+1);
            #pragma omp task final (depth >= MAXDEPTH) shared (res2)
            res2 = rec_processing (data, start+nrows/2, nrows-nrows/2, depth+1);
            #pragma omp taskwait
         }
         else {
            res1 = rec_processing (data, start, nrows/2, depth);
            res2 = rec_processing (data, start+nrows/2, nrows-nrows/2, depth);
         }
    }
    return res1 + res2;
}

int main()
{
  int data[N][N];
  int outofrange;
  ...
  #pragma omp parallel
  #pragma omp single
  outofrange = rec_processing(data, 0, N, 0);
  ...
}
```

A recursive task descomposition with *Tree* strategy was applied to the code. In addition, to control

the task creation overhead, a cutoff is added, which allows the creation of tasks until the recursion level equal to MAXDEPTH is reached.