PAR – In-Term Exam – Course 2024/25-Q1

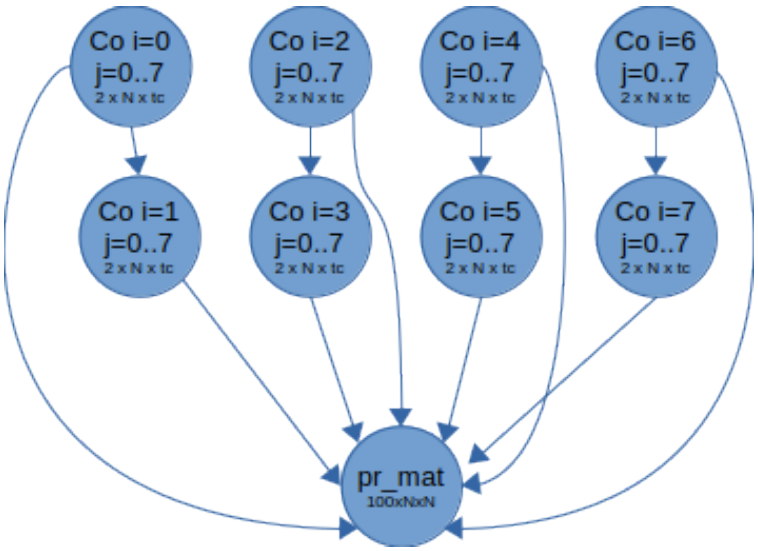
November 6^{th}, 2024

Problem 1 (4.0 points) Given the following code:

```
int mat[N][N];
void main() {
   for (int i=0; i<N; i++) {
     tareador_start_task("compute");
     for (int j=0; j<N; j++) {
        if (i%2 == 0)                          // cost of this line: tc t.u.
          mat[i][j] = mat[i][j] + mat[i+1][j]; // cost of this line: tc t.u.
        else
          mat[i][j] = mat[i][j] - mat[i-1][j]; // cost of this line: tc t.u.
     }
     tareador_end_task("compute");
   }
   tareador_start_task("print_matrix");
   print_matrix(mat);                          // cost of this function: 100*N*N t.u.
   tareador_end_task("print_matrix");
}
```

1. (1 point) Draw the Task Dependence Graph (TDG) based on the above Tareador task definitions and for `N=8`. Each task should be clearly labeled with the values of ranges of i, j, if needed, and its cost in time units (look at comments of the code to figure out the task cost lines).

   Solution:



   There are N `compute` tasks, one for each iteration of the outer loop `i`. The tasks computing even iterations have no Read After Write (RAW) dependencies. However, tasks computing odd iterations depend on the previous iteration. Each of these N `compute` tasks executes the whole set of iterations of the inner loop `j`, i.e. it performs the inner loop body N times. The associated cost of each task is $2 \times N \times t_c$ time units (which results in $16 \times t_c$ t.u. for $N = 8$) regardless of computing an even or odd iteration.

   A final task `print_matrix` prints the whole matrix. It depends on all the previous tasks since each row is previously updated by a different `compute` task. Its associated cost is $100 \times N^2$ time units.

2. (2.0 points) Assume `N=8` and the task costs of previous exercise. Compute the values for T_1, T_∞ and P_{min} and draw the temporal diagram for the execution of the TDG in the previous question on P_{min} processors.

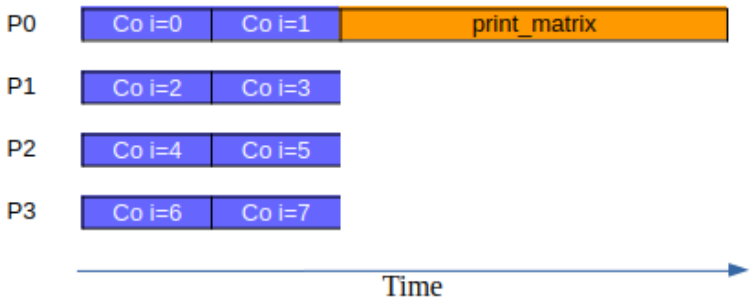
   Solution: Assuming `N=8`,

   $T_1 = N \times 2 \times N \times t_c + 100 \times N^2 = 128 \times t_c + 6400$

   $T_\infty = 2 \times 2 \times N \times t_c + 100 \times N^2 = \ \ 32 \times t_c + 6400$

   $P_{min} = 4$

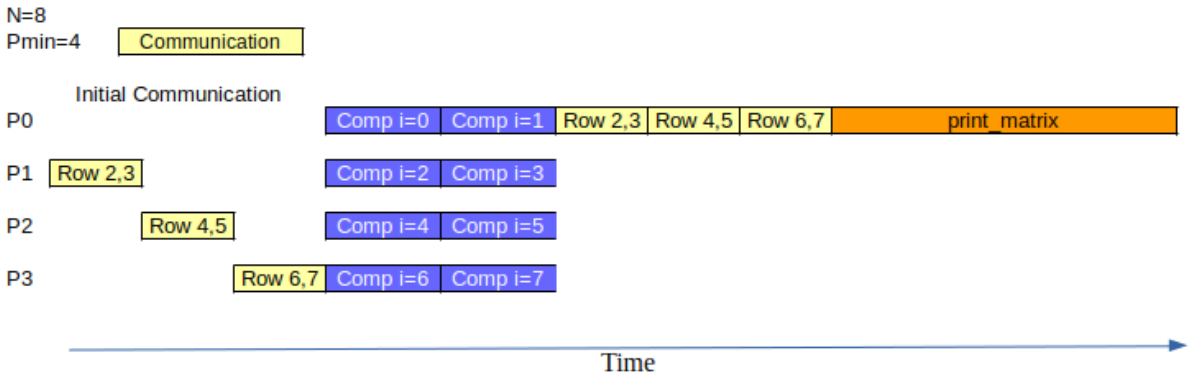
   Temporal diagram for the execution of the TDG in the previous question on $P_{min} = 4$ processors.



3. (1.0 points) Assume the same task definition, `N` a very large value multiple of 2, and consider a distributed memory architecture with $P = P_{min}$ processors. Let's assume that matrix `mat` is initially stored in the memory of processor 0 and tasks are scheduled to P processors in a similar way as you indicated in previous exercise, trying to minimize the data sharing overheads. Write the expression, function of N and $P = P_{min}$, that determines the execution time, T_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:

   Next, we show a temporal diagram for the execution of the tasks in the TDG shown in the 1^{st} question on $P_{min} = 4$ processors, considering the data sharing overheads. This is an example of the execution of the application for $N = 8$, for which $P_{min} = 4$. Note that processors start their computation once the initial communication is done, as explained in class, to symplify the data sharing model.



   In general, $P_{min} = \frac{N}{2}$, where each processor executes the set of two `compute` tasks which compute consecutive even-odd iterations. And one of them executes the final task `print_matrix`.

$T_{Computations} = 2 \times 2 \times N \times t_c + 100 \times N^2$

Prior to performing any computation, $P_1, P_2 \ldots P_{min} - 1$ need to receive 2 consecutive rows from P_0. Consequently, since P_0 can only serve one remote access at a time:

$T_{Initial\_Data\_Sharing} = (P_{min} - 1) \times (t_s + 2N \times t_w)$

Regardless of the processor in which task `print_matrix` is executed, it'll need to receive 2 consecutive rows from each of the other $P_{min} - 1$ processors. Thus,

$T_{Final\_Data\_Sharing} = (P_{min} - 1) \times (t_s + 2N \times t_w)$

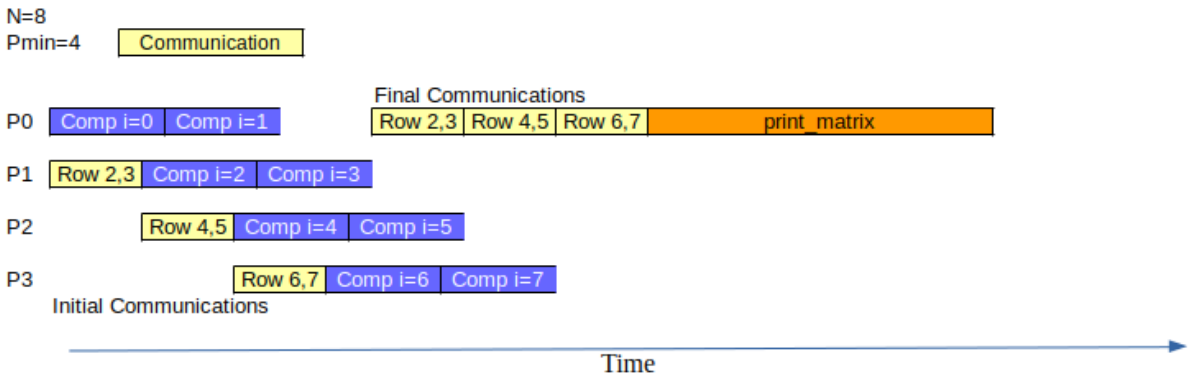
Therefore,

$$T_{P_{min}} = T_{Initial\_Data\_Sharing} + T_{Computations} + T_{Final\_Data\_Sharing}$$
$$= (P_{min} - 1) \times (t_s + 2N \times t_w) + 4 \times N \times t_c + 100 \times N^2 + (P_{min} - 1) \times (t_s + 2N \times t_w)$$

$$= 4 \times N \times t_c + 100 \times N^2 + 2 \times (P_{min} - 1) \times (t_s + 2N \times t_w)$$
$$= 4 \times N \times t_c + 100 \times N^2 + 2 \times (\tfrac{N}{2} - 1) \times (t_s + 2N \times t_w)$$
$$= 4 \times N \times t_c + 100 \times N^2 + (N - 2) \times (t_s + 2N \times t_w)$$

Alternative optimized solution:

If we depart from the simplified model used in class and overlap communications with computations, then for $N = 8$:



And, in general:

$$T_{P_{min}} = T_{Computations} + T_{Data\_Sharing}$$
$$= 4 \times N \times t_c + 100 \times N^2 + (P_{min} - 1) \times (t_s + 2N \times t_w) + 1 \times (t_s + 2N \times t_w)$$
$$= 4 \times N \times t_c + 100 \times N^2 + \tfrac{N}{2} \times (t_s + 2N \times t_w)$$

Problem 2 (3.0 points) Consider the following sequential code:

```
int mat[N][N];

void main() {
    for (int i=0 ;i<N; i++)
      for (int j=0; j<N; j++)
        if (i%2 == 0)
             mat[i][j] = mat[i][j] + mat[i+1][j];
        else
             mat[i][j] = mat[i][j] - mat[i-1][j];
}
```

Assume N is a very large constant multiple of 2. Write an scalable OpenMP parallel program, being P the number of available processors, with the following conditions: 1) You are ONLY allowed to use explicit tasks, but NOT allowed to use taskloop; 2) The granularity of the task corresponds to the process of a chunk of BS consecutive rows of the matrix; 3) The synchronization needed to preserve the same result of the sequential execution, has to be minimized and 4) the overhead due to task creation and execution management has to be minimized.

Solution:

Analyzing the true dependencies in the sequential program we see that each odd row only depends on the previous (even) row. In this way, in order to minimize the synchronization we should guarantee that the chunk of rows assigned to a task should be an even number. In order to minimize the overhead of task management we could create a unique task for processing the maximum number of consecutive rows depending on N and P.

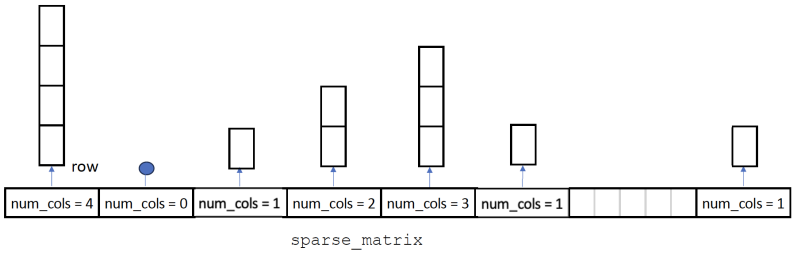
A possible solution:

```
int mat[N][N];

void main() {

 #pragma omp parallel num_threads(P)
 #pragma omp single
 {
    int BS = N/P;
    if (BS%2) BS++;
    for (int ii=0; ii<N; ii+=BS)
    #pragma omp task
    for (int i=ii; i<min(N,ii+BS); i++)
        for (int j=0; j<N; j++)
            if (i%2 == 0)
                mat[i][j] = mat[i][j] + mat[i+1][j];
            else
                mat[i][j] = mat[i][j] - mat[i-1][j];
 } }
```

We also could analyze the execution of each pair of even and odd rows and conclude that they could be processed jointly in the same iteration leading to the following code:

```
#pragma omp parallel num_threads(P)
#pragma omp single
{
    int BS = N/P/2;
    for (int ii=0; ii<N; ii+=BS*2)
    #pragma omp task
    for (i=ii; i<min(N,ii+BS*2); i+=2)
        for (j=0; j<N; j++) {
            int tmp = mat[i][j];
            mat[i][j] = mat[i][j] + mat[i+1][j];
            mat[i+1][j] = - tmp;
        }
```

Problem 3 (3.0 points) Given the following representation of a sparse matrix:



and the following data structures that model the sparse matrix with a sequential recursive algorithm to process it:

```
#define N ... // a large value and represents the number of rows
#define MIN_ROWS 2
#define MIN_NELEMS N
typedef {
    float value;
    int col;
} tRow;
typedef struct {
    int num_cols; // number of columns in the row
    tRow *row;
} tEntry;

// Perform calculation on first n rows of the sparse matrix.
float process (tEntry *sm, int n);

// Calculate pivot index (row index) such that the sum of num_cols (to process)
// up to row index (not included) is **similar** to the sum of num_cols (to process)
// at row index and after. This pivot index is returned in pointer int *index
// This total sum of num_cols before pivot index is also returned in *total_left
void partition (tEntry *sm, int n, int *index, int *total_left);

// Recursive calculation on the first n rows of sparse matrix sm
float rec_process (tEntry *sm, int n, int total_elems) {
    float result;
    if (n <= MIN_ROWS || total_elems <= MIN_NELEMS)
        result = process (sm, n);
    else {
        int index, total_left;
        partition (sm, n, &index, &total_left);
        result =  rec_process (sm, index, total_left);
        result += rec_process (&sm[index], n-index, total_elems-total_left);
    }
    return result;
}
int main() {
    tEntry sparse_matrix[N];
    int total_elems;
    ...
    float result = rec_process (sparse_matrix, N, total_elems);
    ...
}
```

We ask you to: Write a parallel OpenMP parallel version using a recursive task decomposition. Select the most appropriate strategy (tree or leaf) that will maximize the processor utilisation assuming a system with a high number of processors. The implementation should take into account the overhead due to task creation, limiting their creation and ensuring tasks are well-balanced.

Solution:

```
#define CUTOFF X // a value greater or equal than 2N
// Recursive calculation on the first n rows of sparse matrix sm
float rec_process (tEntry *sm, int n, int total_elems) {
    float result1, result2 = 0.0;
    if (n <= MIN_ROWS || total_elems <= MIN_NELEMS) {
        result1 = process (sm, n);
    }
    else {
        int index, total_left;
        partition (sm, n, &index, &total_left);
        if (!omp_in_final()) {
           #pragma omp task shared(result1) final (total_left < CUTOFF)
            result1 =  rec_process (sm, index, total_left);
           #pragma omp task shared(result2) final (total_elems-total_left < CUTOFF)
            result2 += rec_process (&sm[index], n-index, total_elems-total_left);
           #pragma omp taskwait
        }
        else {
            result1 =  rec_process (sm, index, total_left);
            result2 += rec_process (&sm[index], n-index, total_elems-total_left);
        }
    }
    return result1 + result2;
}
int main() {
    tEntry sparse_matrix[N];
    int total_elems;
    ...
    #pragma omp parallel
    #pragma omp single
    float result = rec_process (sparse_matrix, N, total_elems);
    ...
}
```

Notice that one task can achieve the cutoff condition, while the other one not, as the conditions are evaluated using different variables. This is in fact an advantage of marking a task as final individually, instead of performing a global condition.

PAR – In-Term Exam – Course 2023/24-Q2

April 4^{th}, 2024

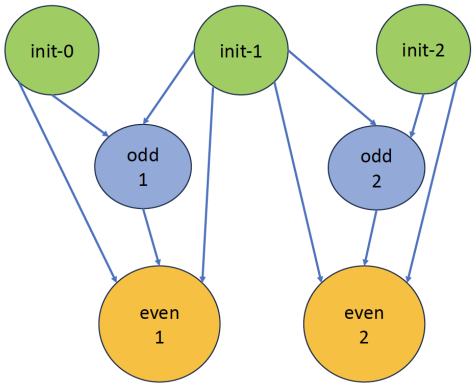
Problem 1 (4.0 points) Given the following code:

```
#define N  ... // value determined at each section
float A[N][N], B[N][N];
...
// initialization
 for (int i=0; i<N; i++) {
   tareador_start_task("init");
   for (k=0; k<N; k++) {
        A[i][k] = init(); // 10 time units
   }
   tareador_end_task("init");
}
// calculation
 for (int i=1; i<N; i++) {
   tareador_start_task("odd"); // Process only odd columns
   for (k=1; k<N; k+=2) {
        B[i][k] = A[i-1][k] + A[i][k] + foo(); // 20 time units
   }
   tareador_end_task("odd");
   tareador_start_task("even"); // Process only even columns
   for (k=2; k<N; k+=2) {
        B[i][k] = A[i-1][k] + A[i][k] + B[i][k-1] + goo(); // 40 time units
   }
   tareador_end_task("even");
 }
```

Assuming that functions `foo`, `goo` and `init` do not modify any element from matrix A and matrix B, **we ask you to:**

1. (1.0 points) Draw the Task Dependence Graph (TDG) based on the Tareador task definitions in the instrumented code above. Each task should be clearly labeled with the value of i and its cost in time units. Assume that $N = 3$.

   Solution:

   Where the cost for each `init` task is 30 time units, for each `odd` task is 20 time units and for each `even` task is 40 time units.

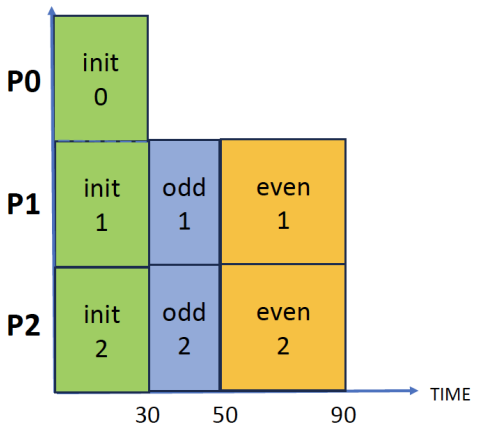
2. (1.0 points) Compute the values for T_1, T_∞ and P_{min}. Draw the temporal diagram for the execution of the TDG if executed using P_{min} processors.

Solution:

$T_1 = 30 \times 3 + (20 + 40) \times 2 = 210$ time units

$T_\infty = 30 + 20 + 40 = 90$ time units

$P_{min} = 3$

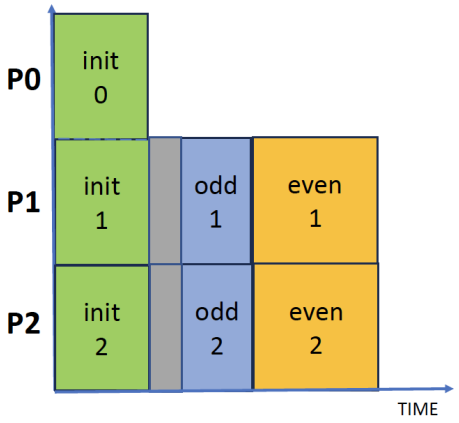


3. (2.0 points) Assume the same task definition, and consider a distributed memory architecture with P processors. Let's assume that matrix A and matrix B are distributed by rows along the P processors (row `i` on processor `i`). Tasks are scheduled on the processor where is allocated the data they have to update. This means that a task that works on row `i` executes on processor `i`.

   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: to answer this question you can assume $N = 3$.

   Solution:



   The grey bars correspond to the data sharing time.

   (b) (1.0 points) Write the expression that determines the execution time T_p as a function of N and P. Identify clearly 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. Remember that at a given time, a processor can only perform one remote access to another processor and serve one remote access from another processor.

Solution:

$T_P = T_{comp} + T_{comm}$

It is necessary to distinguish if N is an odd or an even value:

i. If N is an odd value, the expression for T_{comp} is:
   $T_{comp} = 10 \times N + (20 \times (N-1)/2 + 40 \times (N-1)/2)$
ii. If N is an even value, the expression for T_{comp} is:
   $T_{comp} = 10 \times N + (20 \times N/2 + 40 \times (N-2)/2)$

$T_{comm} = ts + (N-1) \times t_w$

Every task `odd-i` and `even-i` needs 2 rows from matrix `A`. One of the rows is available in the local processor and the other has to be transferred from processor `i-1`, which is the one that counts for the data sharing overhead.

Problem 2 (3.0 points) Consider the following sequential code:

```
object_type *mat[256][256];
int histo[5] = {0, 0, 0, 0, 0};
int object_val(object_type *p); // return value of object *p in the range 0..4

int main() {
     for (int i=0 ; i<256; i++)
        for (int j=0; j<256; j++)
            histo[object_val(mat[i][j])]++;
}
```

that computes the histogram of the values in the range [0..4] related to the objects pointed by a square matrix of 256x256 elements.

We ask you to: Write an iterative task decomposition strategy using OpenMP that efficiently exploits the parallellism reducing task creation and synchronization overheads. You can propose a solution based on implicit or explicit tasks.

Solution with implicit tasks:

```
int main() {
   #pragma omp parallel
   {
    int BS = 256/omp_get_num_threads(); // assume nt divides 256
    int id = omp_get_thread_num();
    int histo_l[5] = {0,0,0,0,0};
    for (int i=id*BS ; i<(id+1)*BS; i++)
       for (int j=0; j<256; j++)
           histo_l[object_val(mat[i][j])]++;

    for (int i=0; i<5; i++)
       #pragma omp atomic
       histo[i] += histo_l[i];
   }
}
```

Solution with explicit tasks:

```
int main() {
   #pragma omp parallel
   #pragma omp single
   {
    int BS = 256/omp_get_num_threads();
    for (int ii=0; ii<256; ii+=BS)
    {
     #pragma omp task
     {
       int histo_l[5] = {0,0,0,0,0};
       for (int i=ii ; i<ii+BS; i++)
         for (int j=0; j<256; j++)
             histo_l[object_val(mat[i][j])]++;

       for (int i=0; i<5; i++)
          #pragma omp atomic
```

```
            histo[i] += histo_l[i];
      }

     }

    }
 }
```

Solution with explicit tasks - taskloop:

```
int main() {
   #pragma omp parallel
   #pragma omp single
   {
    #pragma omp taskloop num_tasks(omp_get_num_threads())
    for (int i=0 ; i<256; i++)
      for (int j=0; j<256; j++)
      {
          int tmp=object_val(mat[i][j]));
          #pragma omp atomic
          histo[tmp]++;
      }
   }
}
```

Problem 3 (3.0 points) Given the following sequential code:

```
unsigned int fibbonacci(unsigned int n) {
    unsigned int fib;
    if(n == 0)       fib = 0;
    else if(n == 1)  fib = 1;
    else {
       fib = fibbonacci(n-1);
       fib += fibbonacci(n-2);
    }
    return fib;
}
int main() {
   unsigned int fibnumber;
   ...
   fibnumber=fibbonacci(N)
   printf("Fibbonacci(%d)\n",fibnumber);
   ...
}
```

We ask you to:

1. (1.5 points) Create a parallel version in OpenMP using a recursive task decomposition for the `fibbonacci` function. Select the most appropriate strategy (*tree* or *leaf*) that will maximize the processor utilisation assuming a system with a high number of processors and without considering task creation and synchronization overheads.
2. (1.0 points) Modify the previous code to implement a task generation control mechanism based on the depth level. Use `MAX_DEPTH` as the maximum depth level to decide if tasks must be created or not.

3. (0.5 points) Assume your first parallel version. Which type of synchronization is required to guarantee that your tasks have finished before the `printf` statement in the main program if you can not assume any implicit or explicit thread barrier between the fibbonacci and `printf` calls?.

Solution a+b+c:

The correct parallel strategy is the tree recursive task decomposition.

The code already modified with the cutoff mechanism follows:

```
unsigned int fibbonacci(unsigned int n, int depth) {
    unsigned int fib1, fib2, fib;
    if(n == 0){
       fib = 0;
    } else if(n == 1) {
       fib = 1;
    } else {
      if (!omp_in_final())
      {
       #pragma omp task shared(fib1) final(depth>=MAX_DEPTH)
       fib1 = fibbonacci(n-1, depth+1);
       #pragma omp task shared(fib2) final(depth>=MAX_DEPTH)
       fib2 = fibbonacci(n-2, depth+1);
       #pragma omp taskwait
      }
      else
      {
       fib1 = fibbonacci(n-1, depth+1);
       fib2 = fibbonacci(n-2, depth+1);
      }
      fib = fib1+fib2;
    }
    return fib;
}

int main() {
   unsigned int fibnumber;
   ...

   #pragma omp parallel
   #pragma omp single
   {
      fibnumber=fibbonacci(N,0)
      // 3) nothing is needed because
      //    each parallel level has a taskwait synch
      printf("Fibbonacci(%d)\n",fibnumber);
   }
   ..
}
```

We do not need any taskwait neither taskgroup since each parallel level already waits for its tasks.

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

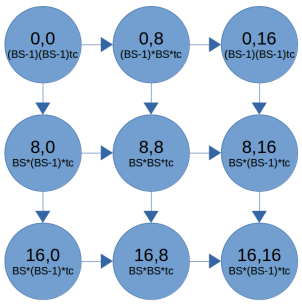
November 2^{nd}, 2023

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:

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

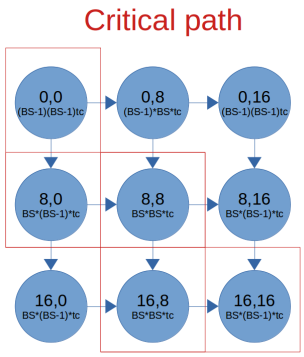
   $T_\infty = ((BS-1)^2 + (BS)(BS-1) + BS^2 + BS^2 + (BS)(BS-1)) \times t_c$

   $P_{min} = 3$

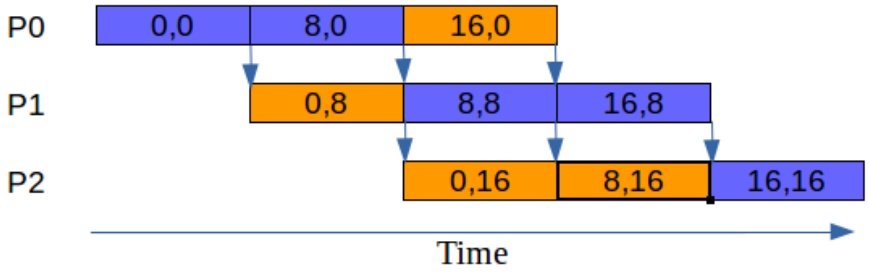
   Considering the particular values of `N` and `BS` provided in the statement of the exercise this translates into:

   $T_1 = 506 \times t_c$

   $T_\infty = 289 \times t_c$



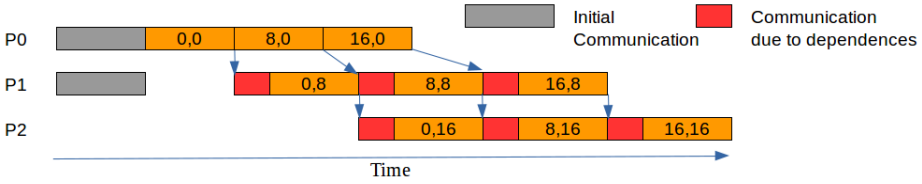
We consider $(BS-1) \approx BS$ in order to simplify the time diagram:



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.

PAR – In-Term Exam – Course 2022/23-Q2

April 26^{th}, 2023

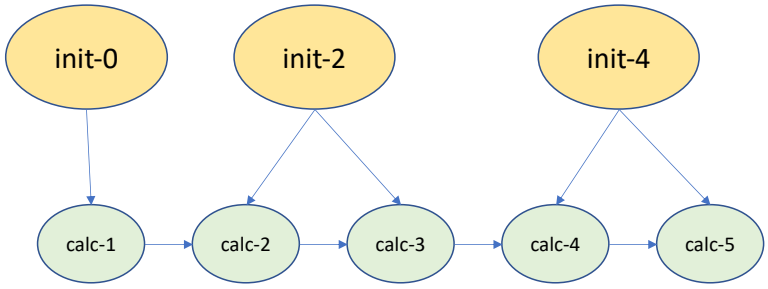
Problem 1 (5.0 points) Given the following code instrumented with *Tareador*:

```
#define N 6  // always a multiple of 2
int m[N][N];
...
// initialization
for (int i=0; i<N-1; i+=2) {   // loop step is 2
   sprintf(label, "init-%d", i);
   tareador_start_task(label);
   for (int k=0; k<N; k++) {
      m[i][k]   = foo(m[i][k], i);
      m[i+1][k] = foo(m[i][k], i);
   }
   tareador_end_task(label);
}

// calculation
for (int i=1; i<N; i++) {  // loop step is 1
   sprintf(label, "calc-%d", i);
   tareador_start_task(label);
   for (int k=0; k<N; k++)
      m[i][k] = goo(m[i-1][k], m[i][k]);
   tareador_end_task(label);
}
```

Assume that the execution time for each `init-i` and `calc-i` task is 20 and 4 time units, respectively, functions `foo()` and `goo()` do not perform any memory access and that the execution cost in time for the rest of the code is negligible. **We ask you to**:

1. (1.0 points) Draw the *Task Dependence Graph* (*TDG*) based on the above *Tareador* task definitions and for $N = 6$. Include for each task its name and its cost in time units. Notice that the outer loop in the initialization phase has a step value of 2.

   Solution:



2. (1.0 points) Compute the values for T_1, T_∞, the amount of *Parallelism* and P_{min}. Draw the temporal diagram for the execution of the *TDG* on P_{min} processors.

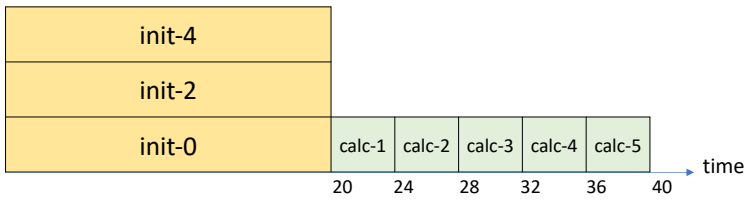
   Solution:

$T_1 = 20 \times 3 + 4 \times 5 = 80$ time units.

$T_\infty = 20 + 4 \times 5 = 40$ time units , determined by the critical path: *init-0, calc-1, calc-2, calc-3, calc-4, calc-5.*

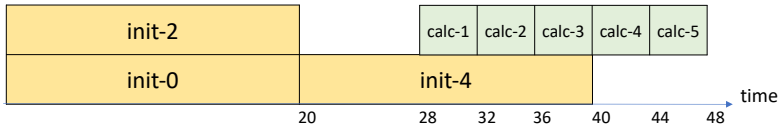
$Par = 80/40 = 2$

$P_{min} = 3$ as shown in the following temporal diagram.



3. (1.0 points) Indicate the best assignment for the tasks in the previous *TDG* to 2 processors that minimizes the computation time. Draw the temporal diagram showing the execution of the *TDG* with the proposed mapping. Compute the values for T_2 and S_2.

   Solution:



   $T_2 = 2 \times 20 + 2 \times 4 = 48$

   $S_2 = T_1/T_2 = 80/48 = 1.67$

4. (2.0 points) Consider a distributed memory architecture with P processors and N any value multiple of 2. Let's assume that matrix m is stored in the memory of processor 0 and that the assignment of task to the P processors is as follows:

   - *init-i* tasks are assigned to `P` different processors (i.e. the number of processors `P` is equal to the number of tasks *init-i*).
   - All *calc-i* tasks are assigned to processor 0

We ask you to: Write the expression that determines the execution time, T_p as a function of N (NOT as a function of P, since P is directly related with the value of N), 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:

Let's write the expression for T_p considering on one hand the contribution from the computation part as T_{comp} and on the other the contribution from data sharing overhead as T_{ov}.

The critical path in the given task allocation to processors is given by the sequence: *init-0, calc-1, calc-2, calc-3, calc-4, calc-5.*

Consequently $T_{comp} = 20 + 4 \times (N - 1)$

The data sharing overhead impacts on T_p : 1) before the execution of all the *init-i* tasks (except for *init-0*), as matrix m is stored in the memory of processor 0; and after that 2) before the execution of all the *calc-i* tasks (except for *calc-0* which use the result data from init-0 on processor 0).

So we obtain from 1) $T_{ov1} = (N/2 - 1) \times (t_s + N \times t_w)$ We only need to access even rows. The odd rows will be generated in the initialization.

and from 2) $T_{ov2} = (N/2 - 1) \times (t_s + 2N \times t_w)$ We need updated even and odd rows.

Finally the expression for T_p is:

$T_p = 20 + 4 \times (N - 1) + (N/2 - 1) \times (t_s + N \times t_w) + (N/2 - 1) \times (t_s + 2N \times t_w)$

Problem 2 (5.0 points) Consider a multiprocessor system with a hybrid NUMA/UMA architecture composed of 2 identical NUMAnodes. Each NUMAnode has 16 Gbytes of main memory and 4 processors, each processor with its own private cache of 16 Mbytes. Memory cache lines are 16 bytes wide and data coherence is guaranteed using a *Write-Invalidate MSI protocol* within each NUMAnode and using a *Write-Invalidate MSU Directory-based* cache coherency protocol between NUMAnodes. Processors 0 to 3 belong to *NUMAnode0* and processors 4 to 7 to *NUMAnode1*. **We ask you to** answer the following two questions:

1. (1.0 point) Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(16 \times 2^{20}) \div 16$ lines, that is 2^{20} lines; therefore the number of bits per cache is $2^{20} \times 2 = 2^{21}$ bits.

2. (1.0 point) Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 2 presence bits for each line of main memory. For the overall 16 GB, this is $(16 \times 2^{30}) \div 16$ lines, that is 2^{30} lines; therefore the number of bits in the directory is $2^{30} \times (2 + 2) = 2^{32}$ bits.

Given the following OpenMP code to be executed on the multiprocessor system described above:

```
#define NUM_THREADS 8
#define M 1
int hist[NUM_THREADS][M];
#pragma omp parallel num_threads(NUM_THREADS)
{
  int id=omp_get_thread_num();
  hist[id][0]=foo();
}
```

and assuming that: 1) no accesses to `hist` are performed before the execution of the parallel region; 2) the initial memory address of `hist` is aligned to the beginning of a memory/cache line and other variables are stored in registers; 3) the size of an `int` data type is 4 bytes; and 4) the Operating System applies the *"First touch"* policy for data allocation in memory. **We ask you to answer the following question:**

3. (0.25 points) How many directory entries will be required to store the coherence information of `hist` and which NUMAnodes will hold them after the execution of the previous parallel region?

   Solution:

   `hist` size (in bytes) is: $NUM\_THREADS * M * 4 = 8 * 1 * 4 = 32$ bytes. Therefore, we require two memory/cache lines of 16 bytes to fit the full `hist` variable; and two directory entries are required to store the coherence information. The NUMAnodes that will hold can be figured out looking at the program. Thread i runs in processor i. So, threads 0 to 3 are running in NUMAnode 0 and any touch to first memory line of `hist` variable (positions `hist[0][0]` to `hist[3][0]`) will make this line to be hold in NUMAnode 0. Threads 4 to 7 are runing in NUMAnode 1 and will make second line of `hist` variable to be hold in NUMAnode 1.

Considering the following execution order for the multiple instances of `hist[id][0]=foo()`: *id=0,2,4,6,1,3,5,7* and the fact that function `foo()` does not perform any memory access to any memory line, **we ask you to:**

4. (1.5 points) Complete the table in the provided answer sheet with the required information in its columns: affected $line_m$, hit or miss in cache, CPU command by processor k ($PrRd_k$/$PrWr_k$), bus transaction(s) by the Snoopy in processor k ($BusRd_k$ / $BusRdX_k$ / $BusUpgr_k$ / $Flush_k$ / $Nothing$), new line state ($I/S/M$) for the copies of $line_m$ in the affected processors, and directory entry information for the affected NUMAnode: number of node, state ($U/S/M$) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0).

After the execution of the previous parallel region processor 0 executes the following sequential code:

```
int sum = 0; // sum is stored in a register
for (int i=0; i<8; i++) // i is stored in a register
  sum += hist[i][0];
```

5. **We ask you to answer the following questions:**
   - (0.25 points) During the execution of the sequential code by processor 0, do you expect any coherence protocol transaction/s between the two NUMANodes?

     Solution:

     Yes. There should be a $RdReq_{0to1}$ and $DReply_{1to0}$ to obtain the second memory line of `hist`, hold in NUMAnode 1.
   - (0.25 points) Once the sequential code has been executed, which will be the home NUMAnode for the memory lines containing `hist` accessed by processor 0?

     Solution:

     Once a memory line is hold in a NUMAnode, its NUMAnode home will remain the same for the full execution. Therefore, no changes happen due to processor 0 accesses, and first and second line of `hist` remain in NUMAnode 0 and 1, respectively.
   - (0.25 points) Which will be the value of the state and presence bits in the directory for those memory lines?

     Solution:

     For the first line of `hist`, the coherence information in the directory entry is the following: state: shared and presence bits: 01, meaning that the memory line is shared with one or more clean copies in NUMAnode 0.

     For the second line of `hist`, the coherence information in the directory entry is the following: state: shared and presence bits: 11, meaning that the memory line is shared with one or more clean copies in NUMAnodes 0 and 1.

And finally,

6. (0.5 points) Do you observe any potential efficiency problem in the parallel region related to the cache coherence protocol? Briefly justify your answer and indicate a modification of the code that avoids this problem.

   Solution: There is false sharing accessing to `hist` variable. Threads are updating consecutive positions of a memory line. The modification should make each position of `hist[i][0]` to be in different memory lines. That can be done by making M constant to be 4. M equal 4 makes that each row of `hist` requires 16 bytes, which is the size of a full memory line, and then, accesses to `hist[i][0]` do not share memory lines.

Student name: .....

Answer for question 2.4

Time Unit		Affected $line_m$	hit or miss	Processor command	Bus transaction/s	Processor : Cache line state	Directory entry		
							NUMAnode #	State	Presence bits
1	hist[0][0]=foo()								
2	hist[2][0]=foo()								
3	hist[4][0]=foo()								
4	hist[6][0]=foo()								
5	hist[1][0]=foo()								
6	hist[3][0]=foo()								
7	hist[5][0]=foo()								
8	hist[7][0]=foo()								

Student name: .....

Answer for question 2.4

Time Unit		Affected $line_m$	hit or miss	Processor command	Bus transaction/s	Processor : Cache line state	Directory entry		
							NUMAnode #	State	Presence bits
1	hist[0][0]=foo()	0	miss	$PrWr_0$	$BusRdX_0$	0:M	0	M	01
2	hist[2][0]=foo()	0	miss	$PrWr_2$	$BusRdX_2$ $Flush_0$	2:M 0:I	0	M	01
3	hist[4][0]=foo()	1	miss	$PrWr_4$	$BusRdX_4$	4:M	1	M	10
4	hist[6][0]=foo()	1	miss	$PrWr_6$	$BusRdX_6$ $Flush_4$	6:M 4:I	1	M	10
5	hist[1][0]=foo()	0	miss	$PrWr_1$	$BusRdX_1$ $Flush_2$	1:M 2:I	0	M	01
6	hist[3][0]=foo()	0	miss	$PrWr_3$	$BusRdX_3$ $Flush_1$	3:M 1:I	0	M	01
7	hist[5][0]=foo()	1	miss	$PrWr_5$	$BusRdX_5$ $Flush_6$	5:M 6:I	1	M	10
8	hist[7][0]=foo()	1	miss	$PrWr_7$	$BusRdX_7$ $Flush_5$	7:M 5:I	1	M	10

PAR – In-Term Exam – Course 2022/23-Q1

November 3^{rd}, 2022

Problem 1 (3.0 points) Given the following code:

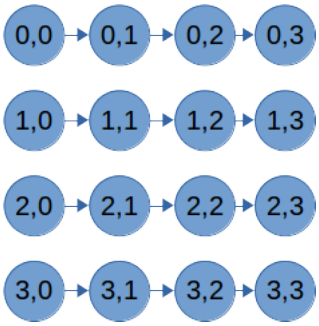
```
#define N  256
#define BS  64
int m[N][N];

for (int ii=0; ii<N/BS; ii++) {
   for (int jj=0; jj<N/BS; jj++) {
      tareador_start_task("task");

      // In general, a task processes a block of BSxBS elements
      // However, if the task is labeled with (ii,jj=0),
      //          this task processes BSx(BS-1) elements
      int i_start = ii*BS; int i_end = i_start+BS;
      int j_start = jj*BS; int j_end = j_start+BS;
      for (int i=i_start; i<i_end; i++)
        for (int j=max(j_start,1); j<j_end; j++)
          m[i][j] = compute (m[i][j], m[i][j-1]);  // tc t.u. (time units)

      tareador_end_task("task");
} }
```

Assume the innermost loop body takes t_c t.u., all variables but matrix m are in registers, function `compute` only reads the values received as arguments and does not modify other positions in the memory, BS perfectly divides N, being BS and N defined in the code above. **We ask you:**

1. (1.0 points) Draw the task dependence graph (TDG), indicating the cost of each task as a function of BS and t_c. Label each task with the values of ii and jj.

   Solution:

   Tasks (ii,jj) when jj!=0 have a cost of $BS \times BS \times t_c$. Tasks (ii,jj) when jj=0 have a cost $BS \times (BS-1) \times t_c$



2. (1.0 points) Compute T_1, T_∞, P_{min} as a function of BS, N and t_c.

   Solution:

   T_1 can be computed as:

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

   T_∞ is the execution time of any of the rows of tasks of the TDG:

   $T_\infty = (BS-1) \times (BS) \times t_c + (\frac{N}{BS} - 1) \times (BS^2) \times t_c$;

This T_∞ is obtained when one processor is assigned to the computation of a row of tasks in the TDG. Therefore:

$P_{min} = \frac{N}{BS}$;

3. (1.0 points) Assuming the assignment of tasks to 4 processors of the table below, calculate T_4 and speedup S_4.

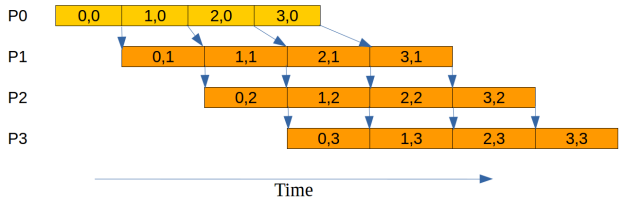
Processor	Tasks
P_0	$(0,0),(1,0),(2,0),(3,0)$
P_1	$(0,1),(1,1),(2,1),(3,1)$
P_2	$(0,2),(1,2),(2,2),(3,2)$
P_3	$(0,3),(1,3),(2,3),(3,3)$

Solution:

$S_4 = \frac{T_1}{T_4}$

$T_1 = N \times (N-1) \times t_c = 65280 \times t_c$ t.u.

Figure below shows the execution timeline considering the assignment of tasks to 4 processors above and their dependences. Each processor executes $\frac{N}{BS}$ tasks. Let's identify those that originate the critical path in the execution timeline. Processor 0 executes tasks with cost $BS \times (BS-1) \times t_c$. First task of Processor 0 contributes to T_4: $((BS-1) \times BS \times t_c)$. Then, first tasks of processors 1 an 2 also contribute to T_4: $((\frac{N}{BS} - 2) \times BS^2 \times t_c)$. Finally, all tasks executed in processor 3 contribute to T_4: $(\frac{N}{BS} \times BS^2 \times t_c)$.



$T_4 = ((BS-1) \times BS + (\frac{N}{BS} - 2) \times BS^2 + \frac{N}{BS} \times BS^2) \times t_c = 28608 \times t_c$ t.u.

$S_4 = \frac{T_1}{T_4} = 2.28\times$

Problem 2 (2.0 points) Given the same code and mapping of tasks to 4 processors as in the previous exercise, assume

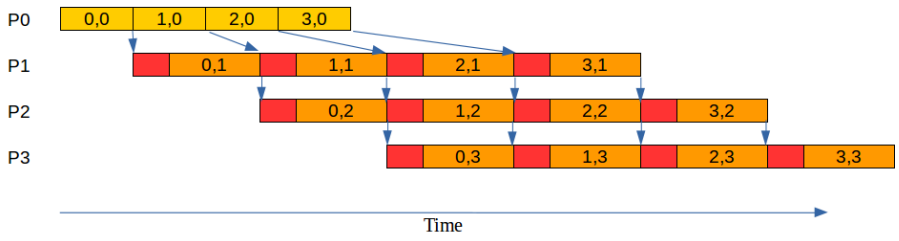
- A distributed-memory architecture with $P = 4$ processors;
- Matrix m is initially distributed by columns (BS consecutive columns per processor).
- Data sharing model with $t_{comm} = t_s + W \times t_w$, being W the number of elements to transfer, and t_s and t_w the start-up time and transfer time of one element, respectively;
- The execution time for a single iteration of the innermost loop body takes t_c t.u.

We ask you: Draw the execution timeline of the execution of tasks and write the expression that determines the execution time T_P, clearly indicating the contribution of the computation time $T_{P\,comp}$ and data sharing overhead $T_{P\,mov}$, as a function of N, BS, P, t_c, t_s and t_w.

Solution:

Figure below shows the execution timeline considering the assignment of tasks to 4 processors above, their dependences and the remote memory accesses. Processors 1, 2 and 3 have to wait for completness of tasks $ii, jj-1$ whose results are saved in the remote memory of the previous processor. As we are now considering data sharing overheads, processors 1, 2 and 3 have to perform a remote memory access of the BS elements of the left boundary before each task is executed. Therefore, the cost of each communication is $t_s + BS \times t_w$. Tasks in processor 0 perform local memory access with no data sharing overhead.

Each processor executes $\frac{N}{BS}$ tasks.



In particular, the contribution of computation and data sharing overheads is the following:

- Processor 0 executes tasks with cost $BS \times (BS-1) \times t_c$. First task of Processor 0 contributes to T_4: $(BS-1) \times BS \times t_c$.
- First tasks of processors 1 an 2 also contribute to T_4: $(\frac{N}{BS} - 2) \times ((t_s + BS \times t_w) + (BS^2 \times t_c))$.
- Finally, all tasks executed in processor 3 contribute to T_4: $(\frac{N}{BS}) \times ((t_s + BS \times t_w) + (BS^2 \times t_c))$.

$T_4 = T_{comp} + T_{comm}$, being:

$T_{comp} = ((BS-1) \times BS + (\frac{N}{BS} - 2) \times BS^2 + \frac{N}{BS} \times BS^2) \times t_c$

$T_{comm} = (\frac{N}{BS} - 2) \times (t_s + BS \times t_w) + \frac{N}{BS} \times (t_s + BS \times t_w) = (2 \times \frac{N}{BS} - 2) \times (t_s + BS \times t_w)$

Problem 3 (5.0 points) Consider a multiprocessor system with a hybrid NUMA/UMA architecture which is composed by 3 identical NUMAnodes. Each NUMAnode has 20 Gbytes of main memory and 2 processors with its own private cache of 8 Mbytes. The memory cache lines are 32 bytes wide, and data coherence is guaranteed using *Write-Invalidate MSI protocol* within each NUMAnode and using a *Write-Invalidate MSU Directory-based* cache coherency protocol among NUMAnodes.

We ask you to answer the following questions:

1. (1.0 points) Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(8 \times 2^{20}) \div 32$ lines, that is 2^{18} lines; therefore the number of bits per cache is $2^{18} \times 2 = 2^{19}$ bits.

2. (1.0 points) Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence, indicating the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 3 presence bits for each line of main memory. For the overall 20 GB, this is $(20 \times 2^{30}) \div 32$ lines, that is 20×2^{25} lines; therefore the number of bits in the directory is $20 \times 2^{25} \times (2 + 3) = 100 \times 2^{25}$ bits.

Given the following OpenMP code excerpt which is executed on processor 0 from the previous described multiprocessor system:

```
#define N (6*1024)
#define NUM_THREADS 6
int v[N], count[NUM_THREADS];
...
for (int k = 0; k < NUM_THREADS; k++)
   count[k]=0;

/*** POINT A ***/

#pragma omp parallel num_threads (NUM_THREADS)
{
   int id=omp_get_thread_num();
   int num_iter = N/NUM_THREADS;

   for (int k = id*num_iter; k < id*num_iter+num_iter; k++) {
       int value = v[k];
       if (is_prime(value)) /* returns true if "value" is a prime number */
           count[id]++;
   }
}
```

and assuming that: 1) the initial memory address of vector `count` is aligned to the start of a memory/cache line; 2) the size of an `int` data type is 4 bytes; and 3) processors 0 and 1 belong to NUMAnode0, processors 2 and 3 belong to NUMAnode1 and processors 4 and 5 belong to NUMAnode2; and 4) the Operating System applies the "first touch" policy for data allocation in memory. **We ask you to:**

3. (1.25 points) Complete the table in the provided answer sheet with the required information: affected cache line (numbered from the first position where vector `count` is allocated), cache line states (`I/S/M`) in processors 0 to 5, directory entry state (`U/S/M`) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0), to keep cache coherence, **when the execution of the previous code reaches POINT A**.

   Solution: The cache line size is 32 bytes, and each element of the vector occupies 4 bytes, so the entire vector `count` fits in a unique memory line.

	Affected line	Home NUMAnode	Cache line state						Directory entry	
			0	1	2	3	4	5	State	Presence bits
count[0]	0	0	M	-	-	-	-	-	M	001
count[1]	0	0	M	-	-	-	-	-	M	001
count[2]	0	0	M	-	-	-	-	-	M	001
count[3]	0	0	M	-	-	-	-	-	M	001
count[4]	0	0	M	-	-	-	-	-	M	001
count[5]	0	0	M	-	-	-	-	-	M	001

4. (1.25 points) Complete the table in the provided answer sheet with the required information: affected cache line, access in cache (hit/miss), CPU command for processor k ($PrRd_k/PrWr_k$), Bus transaction(s) from Snoopy in processor k ($BusRd_k$ / $BusRdX_k$ / $BusUpgr_k$ / $Flush_k$ / $Nothing$), cache line states in processors 0 to 5, directory entry state and presence bits, to keep cache coherence, **AFTER the execution of each** memory access in the table. Assume initially memory and cache states from previous question.

   Solution:

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state						Directory entry	
					0	1	2	3	4	5	State	Presence bits
Processor 1 reads count[1]	0	Miss	PrRd1	BusRd1 / Flush0	S	S	-	-	-	-	S	001
Processor 0 reads count[0]	0	Hit	PrRd0	Nothing	S	S	-	-	-	-	S	001
Processor 1 writes count[1]	0	Hit	PrWr1	BusUpgr1	I	M	-	-	-	-	M	001
Processor 2 reads count[2]	0	Miss	PrRd2	BusRd2 / Flush1	I	S	S	-	-	-	S	011
Processor 0 writes count[0]	0	Miss	PrWr0	BusRdX0	M	I	I	-	-	-	M	001

   Notice that, as the entirely vector `count` fits in a cache line, access to any element of the vector provokes access to the same cache line.

5. (0.5 points) Have you observed any potential efficiency problem in the previous code? Justify briefly your answer.

   Solution:

   False sharing: Vector `count` lies on a single memory line, then all the threads are eventually accessing for writing the same cache line during the concurrent execution of the program, but to different memory addresses, resulting in unnecessary coherence traffic, even between nodes.

   Possible memory bottleneck in the access to memory of processor 0: Vector count is entirely allocated in that memory, and all the threads have to access to it during execution.

Student name: .....

Answer for question 3.3

	Affected line	Home NUMAnode	Cache line state						Directory entry	
			0	1	2	3	4	5	State	Presence bits
count[0]										
count[1]										
count[2]										
count[3]										
count[4]										
count[5]										

Answer for question 3.4

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state						Directory entry	
					0	1	2	3	4	5	State	Presence bits
Processor 1 reads count[1]												
Processor 0 reads count[0]												
Processor 1 writes count[1]			.									
Processor 2 reads count[2]												
Processor 0 writes count[0]												

PAR – In-Term Exam – Course 2021/22-Q2

April 6^{th}, 2022

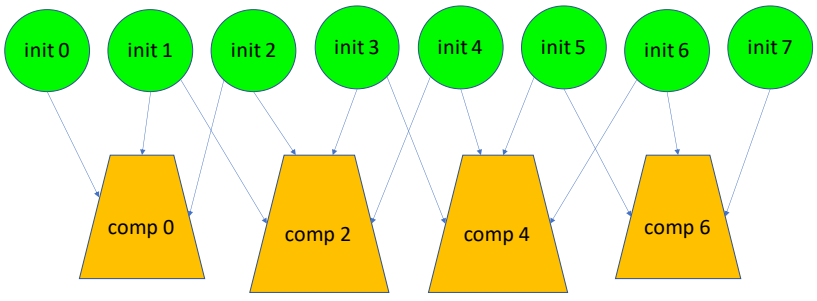
Problem 1 (4 points) Given the following code using *Tareador* for task annotations:

```
#define R  8
#define BS 2
int x[R][N], y[R][N];
...
// initialization phase
for (int i=0; i<R; i++) {
    tareador_start_task ("init");
    for (int k=0; k<N; k++)
        x[i][k] = foo (i); // no other memory accesses inside foo
    tareador_end_task ("init");
}
// computation phase
for (int i=0; i<R; i+=BS) {
    tareador_start_task ("comp");
    for (int ii=max(1,i); ii<min(R-1,i+BS); ii++)
        for (int k=0; k<N; k++)
            y[ii][k] = compute (x[ii][k], x[ii-1][k], x[ii+1][k]);
                       // no other memory accesses inside compute
    tareador_end_task ("comp");
}
```

Assume that the execution time for functions `foo` and `compute` is 2 and 20 time units, respectively, and that the execution cost in time for the rest of the code is negligible. **We ask you to**:

1. Draw the *Task Dependence Graph* (TDG) based on the above *Tareador* task definitions. Include for each task its name followed by the iteration number that generates it and its cost in time units (as a function of `N`).

   Solution:



Where task costs are:
cost (*init i*) = 2 x N where *i=0..7*
cost (*comp 0*) = cost (*comp 6*) = 20 x 1 x N
cost (*comp 2*) = cost(*comp 4*) = 20 x 2 x N

2. Calculate the values for T_1, T_∞ and amount of *Parallelism* (as a function of `N`).

   Solution:

   $T_1 = 8 \times 2 \times N + 2 \times 20 \times 1 \times N + 2 \times 40 \times N = 136 \times N$

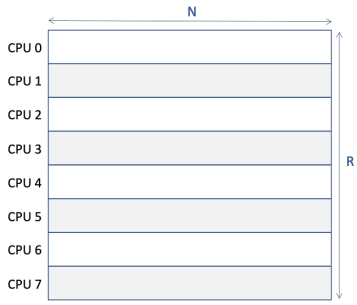
   $T_\infty = 2 \times N + 40 \times N = 42 \times N$, determined by any of the critical paths going through either `comp 2` or `comp 4`. For example, `init 2`–`comp 2`.

   $Parallelism = (136 \times N)/(42 \times N) = 3.2$.

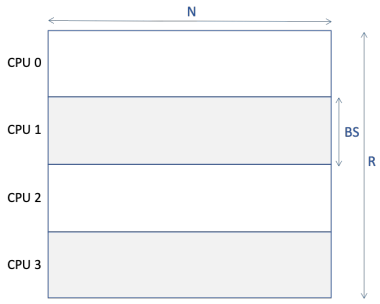
   Although it is not asked in the problem, observe that in this case $P_{min} = 6$. Why?

3. Given the following task allocation for $P = 8$ processors (CPUs) and mapping of matrices `x` and `y` by rows to processors ($R = 8$ and $BS = 2$):

Processor	Task assignment
0	init 0, comp 0
1	init 1, comp 2
2	init 2, comp 4
3	init 3, comp 6
4	init 4
5	init 5
6	init 6
7	init 7



Matrix x mapped to 8 processors



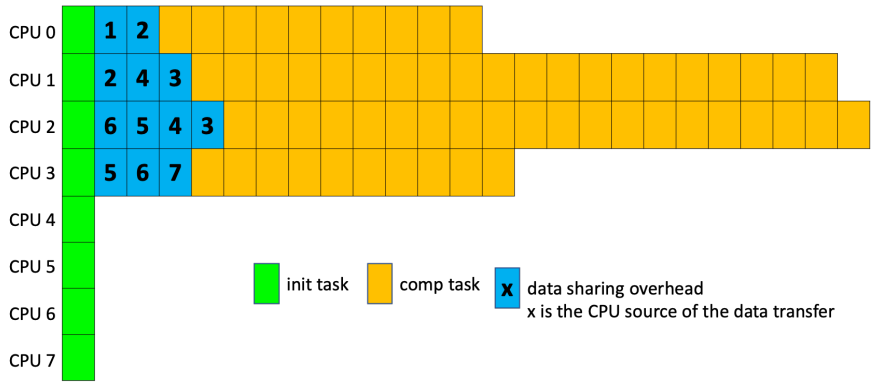
Matrix y mapped to 4 processors

Write the expression that determines the execution time, T_8 as a function of `N`, 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 any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Solution:

First let's find out the critical path considering data sharing overheads. Observe that `comp 0` executed by processor 0 has to perform two remote accesses (to get the row initialized in `init 1` by processor 1 and to get the row initialized in `init 2` by processor 2); `comp 2` executed by processor 1 has to perform three remote accesses (to get the row initialized in `init 2` by processor 2, the row initialized in `init 3` by processor 3 and the row initialized in `init 4` by processor 4); `comp 4` executed by processor 2 has to perform four remote accesses (to get the row initialized in `init 3` by processor 3, the row initialized in `init 4` by processor 4, the row initialized in `init 5` by processor 5, and the row initialized in `init 6` by processor 6); and finally, `comp 6` executed by processor 3 has to perform three remote accesses (to get the row initialized in `init 5` by processor 5, the row initialized in `init 6` by processor 6 and the row initialized in `init 7` by processor 7). Therefore it is clear that the critical path is determined by processor 3 executing `init 2` and `comp 4`,

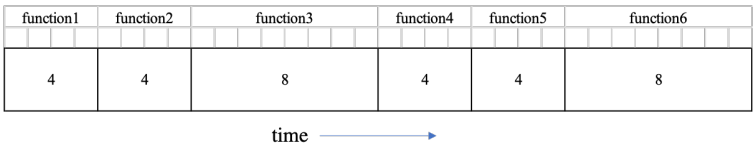
The following temporal diagram shows the execution taking into account the data sharing overheads. There is also indicated a possible scheduling of data transfers to show that it is possible to make it.



The data sharing overhead cost is given by the expression: $T_8^{datasharing} = 4 \times (t_s + t_w \times N)$

so $T_8 = T_8^{comp} + T_8^{datasharing} = (2 \times N + 40 \times N) + (4 \times (t_s + t_w \times N))$

Problem 2 (2 points) The following figure shows the timing diagram for the sequential execution of an application on 1 processor:



The figure has a set of rectangles, each rectangle representing the serial execution of a function with its associated cost in time units. In a first attempt the programmer has been able to parallelize functions 1, 2, 3, 4 and 5, which can be ideally decomposed; **function 6 remains sequential. We ask you to answer the following questions**:

1. What is the parallel fraction ϕ?

   Solution: $\phi = \frac{4+4+8+4+4}{4+4+8+4+4+8} = \frac{24}{32} = \frac{3}{4}$

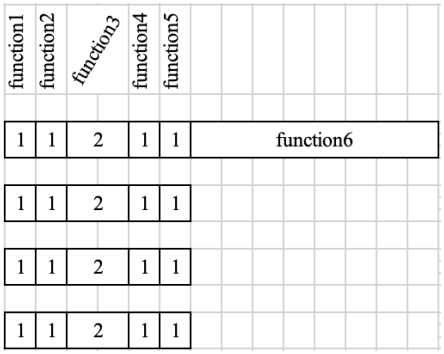
2. Which would be the maximum speedup that you could achieve based on Amdahl's law ($S_{p\to\infty}$)?

   Solution: $S_{p\to\infty} = \frac{1}{(1-\phi)} = 4$

3. Draw the time diagram of an ideal parallel execution of the application to obtain maximum speed-up with 4 processors. Which is the value for S_4 that is achieved?

   Solution:

   The proposed parallel execution is shown below, giving the ideal speed–up for 4 processors: $S_4 = \frac{T_1}{T_4} = \frac{32}{14} = \frac{16}{7} = 2.28$.



In a second attempt the programmer has been able to fully parallelize all funtions in the application (total workload of 32 time units), with ideal scalability. Please **answer the following two questions:**

4. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and strong scaling?

   Solution: $work\_load\_per\_thread = \frac{32}{8} = 4$ time units.

5. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and weak scaling?

   Solution: $work\_load\_per\_thread = 32$ time units.

Problem 3 (4 points) Given the following code excerpt, including OpenMP directives, to be executed on a UMA architecture with 8 processors, each one with a 1 MB fully-associative cache memory, and cache coherence maintained with the simplest write–invalidate MSI protocol explained in class, sharing the access to 16 GB of main memory:

```
#define NUM_THREADS 8
#define N 262144                 // N multiple of NUM_THREADS

int a[N], b[N];
struct {
    int zeros[NUM_THREADS];
    int positives;
} count;

#pragma omp parallel num_threads(NUM_THREADS)
{
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0)
            #pragma omp atomic
            count.positives++;
    }
}
```

Observe that each implicit task executes a chunk of `num_elems` consecutive iterations of the loop. Assume that processor i executes implicit task with $id = i$. Also assume that each integer (`int`) variable occupies 4 bytes, a memory (cache) line occupies 32 bytes, and that the initial address of vectors `a` and `b` as well as structure `count` are aligned with the start of a cache line. All other variables are stored in registers (not in memory). Processors have no cached copies of `a`, `b` and `count` in their private cache memories before starting the execution of the parallel region. We ask you to **answer the following questions:**

1. How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to **vectors `a` and `b`**.

   Solution: Each vector `a` and `b` occupies (262144 elements * 4 bytes/element) = 1 MB. Since elements are equally distributed among the 8 processors, each local cache will need 128 KB for each vector, which correspond to 128 KB / 32 bytes/line = 4096 lines. Therefore each processor will place on the bus 4096 *BusRd* command to read its elements of `a` and the same for `b`; since `a` is also written, 4096 *BusUpgr* commands (not *BusRdX* since the access results in a cache hit) will also be placed on the bus from each processor. Both vectors fit in the caches, so no *Flush* commands are generated for the lines of vector `a`. In total: 8192 BusRd and 4096 BusUpgr, no BusRdX and no Flush.

2. How many memory lines does **structure `count`** occupy? Does the access to `count` in the program causes *true* and/or *false* sharing? Briefly reason your answer.

   Solution: Structure `count` occupies (8 + 1) int * 4 bytes/int = 36 bytes. Therefore it occupies 2 cache lines. The access to `count.positives` causes *true* sharing, since that field is updated by all processors whenever a positive value is computed for `a`. The access to `count.zeros` causes *false* sharing, since each processor is accessing to a different element of the vector `zeros` but all of them reside in the same cache line.

3. In order to reduce the coherence traffic that is caused by the accesses to **vector `count.zeros`** the programmer has changed the definition of `count` and the access to it, as follows:

```
struct {
    int zeros[NUM_THREADS*PADDING];
    int positives;
} count;
...
        if (a[i] == 0) count.zeros[id*PADDING]++;
...
```

Which would be the most appropriate value for constant `PADDING`? How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to `count.zeros`?

Solution: Since this access is causing a *false* sharing problem, coherence traffic can only be eliminated if we use padding, so that each element of vector `zeros` occupies a different memory line. We achieve this with `PADDING=8`, so that the elements accessed by different processors are $8 \times 4 = 32$ bytes apart. With this each processor will place one *BusRd* and one *BusUpgr* command in the shared bus during the whole execution of the parallel region.

If the multiprocessor architecture is upgraded to a NUMA system with 8 nodes, each node with a single processor, a private cache memory of 1 MB, and a portion of main memory of 2 GB, and all variables (vectors `a` and `b` and structure `count`) are physically mapped to NUMA node 0 by the operating system (first touch policy) before starting the execution of the parallel region, with no cached copies in any of the nodes. We ask you to **answer the following questions:**

4. How many entries in the directory **of each NUMA node** will be used to store the coherence information for **vectors `a` and `b`**? After the execution of the parallel region, how many bits in the *sharers list* of each of these directory entries will be set to 1 and in which state will those memory lines be?

   Solution: As calculated before, each vector `a` and `b` occupies (262144 elements * 4 bytes/element)/32 bytes/line = 32768 lines. Since all elements are mapped in NUMA node 0, 32768*2=65536 entries will be used in that node. No entries will be used in the rest of nodes. Since each line is only accessed by the processor of a single NUMA node, only one bit in the *sharers list* will be set to one, with M state for vector `a` and S state for vector `b`.

5. After executing the program the programmer realized that the access to `count.positives` was creating a performance bottleneck, so she/he proposed the following program transformation for the parallel region, making use of a per–thread copy `tmppos` of the shared variable `count.positives`, which is accumulated into it before exiting the parallel region:

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    int tmppos = 0;
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0) tmppos++;
    }
    #pragma omp atomic
    count.positives = count.positives + tmppos;
}
```

Assume that the processor in NUMA node 0 is the first executing `#pragma omp atomic` and that its execution ensures read/write atomicity in the access to `count.positives` (i.e. while one processor is accessing to it inside the `pragma` no other processors will be able to access it). Which of the following statements is/are true? **(selected wrong statements penalize the mark that you can obtain in this question)**

(a) The proposed transformation can never improve performance since the number of of positives found in vector `a` does not change.

(b) If the number of positive results in vector `a` is much larger than 1, the ONLY improvement in performance comes from the fact that `#pragma omp atomic` has been removed from the loop body.

(c) The last processor x executing `#pragma omp atomic` will first send a $RdReq_{x \to 0}$ command, which will provoke a $Fetch_{0 \to y}$ command to the remote node y that performed the previous update. As a consequence, $Dreply_{y \to 0}$ and $Dreply_{0 \to x}$ will be generated in order to get the updated line in processor x. The line is also updated in main memory of node 0.

(d) After that, the same processor x will send an $UpgrReq_{x \to 0}$ to change the state of the line in the home node and an $Invalidate_{x \to y}$ command to invalidate the copy in cache of node y, which will reply with an $Ack_{y \to x}$ to notify the completion of the command.

(e) At the end of the parallel region the directory entry associated to the memory line containing `count.positives` will be in state M with only the bit associated to node x in the sharers list active to 1.

Note: *$Command_{a \to b}$ refers to a NUMA command sent from node a to node b.*

Solution:

{False, False, True, False, True}

PAR – In-Term Exam – Course 2021/22-Q1

November 8^{th}, 2021

Problem 1 (2 points)

We have a sequential code that we want to parallelize. Our code has four disjoint parts executed one after the other, namely *Start*, *Init*, *Compute* and *End*, which take 1, 10, 100 and 2 time units, respectively.

1. If we only parallelize the *Compute* phase, which would be the value for the parallel fraction ϕ? Assuming that *Compute* can be perfectly parallelized, and there are no overheads resulting from its parallelization, which would be the value for the ideal speed–up $S_{p\to\infty}$?

   Solution:

   $\phi = \frac{100}{113} = 0.885$; $1 - \phi = 0.115$

   Then, according to Amdahl's law when the number of processors $P \to \infty$:

   $S_{p\to\infty} = \frac{1}{1-\phi} = \frac{1}{0.115} = 8.69$

2. Our system, however, has a parallelization overhead proportional to the number of processors being used. Which would be the value for the speed–up when using 10 processors (S_{10}) if the parallelization of *Compute* can be perfectly parallelized but incurs in an overhead of 0.001 time units per processor being used?

   Solution:

   Considering the overheads and Amdahl's law:

   $S_P = \frac{1}{1-\phi+\phi/P+0.001*P/T_1}$

   Thus,

   $S_{10} = \frac{1}{1-\phi+\phi/10+0.001*10/T_1} = 4.91$

   or, alternatively, just applying the definition of $S_p = T_1/T_p$:

   $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{13+100/10+0.001*10} = \frac{113}{23.01} = 4.91$

3. Next, we parallelize the *Init* phase on two processors. Regrettably, in this *Init* phase the parallelization cannot be scaled beyond two processors. Assuming again that there are no overheads due to the parallelization, which would be the new value for S_{10} in this case?

   Solution:

   Since a part of the code cannot be perfectly parallelized we cannot use Amdahl's law here. Thus, our only choice in this case is using the definition $S_P = \frac{T_1}{T_P}$.

   $T_{10} = 1 + 5 + 10 + 2 = 18$

   Therefore: $S_{10} = \frac{T_1}{T_{10}} = \frac{113}{18} = 6.277$

Problem 2 (2.5 points)

Assume a program composed of two parallel regions: *Region A* and *Region B*; both regions scale ideally when executed with P processors. There is no code outside these two regions in the program. The first region, *Region A*, is basically a double nested loop reading matrix `Matrix_a` and writing into matrix `Matrix_b`:

```
for (int row = 0; row < N; row++)
  for (int col = 0; col < N; col++)
         Matrix_b[row][col] = foo(Matrix_a[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `row` loop, with tasks computing blocks of rows assigned to processors in ascending order. Once the execution of *Region A* is finished, the program proceeds with *Region B*, which is also a double nested loop reading matrix `Matrix_b` and writing into matrix `Matrix_c`:

```
for (int col = 0; col < N; col++)
  for (int row = 0; row < N; row++)
        Matrix_c[row][col] = goo(Matrix_b[row][col]);
```

For this region the programmer has implemented a task decomposition in which each task computes N/P consecutive iterations of the `col` loop, again with tasks computing blocks of columns assigned to processors in ascending order. After the execution of *Region B* the program terminates.

The three matrices have N rows and N columns, but are distributed in three different ways: `Matrix_a` is totally stored in the memory of processor 0; `Matrix_b` is distributed by rows among all P processors, so that each processor stores N/P consecutive rows in its memory (blocks of rows mapped to processors in ascending order); and `Matrix_c` is distributed by columns among all P processors, so that each processor stores N/P consecutive columns in its memory (blocks of columns mapped to processors in ascending order).

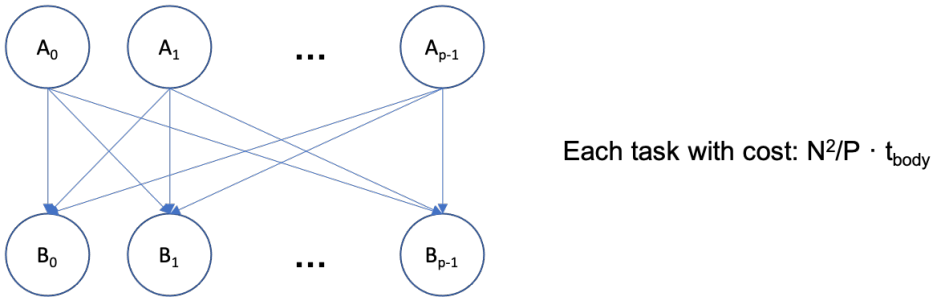
You can assume the data sharing model explained in class in which the overhead to perform a remote access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred. At any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor. You can also assume that the execution of body of each innermost loop takes t_{body}.

We ask you to:

1. Draw the *Task Dependence Graph* (TDG) for the program described above, indicating the cost of each task.

   Solution:

   The TDG includes P nodes, each with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region A* and P nodes, each also with a cost of $\frac{N^2}{P} \times t_{body}$, for *Region B*. Each task in *Region B* depends on all tasks in *Region A*, as shown in the TDG below.



2. Identify which remote accesses have to be performed during the execution of the parallel program, clearly identifying the processors involved in each remote access, the number of elements that need to be transferred and when the remote accesses should occur.

   Solution:

   Before the computation in *Region A* can start, each processor should access to those elements of `Matrix_a` that are needed, that is: each processor from 1 to $P-1$ should access to N/P consecutive rows (each one with N elements) from processor 0. Since this processor can only serve one remote request at a time, all these remote accesses are sequentialised. Before the computation in *Region B* can start, each processor should access to those elements of `Matrix_b` that are needed, that is: each processor (from 0 to $P-1$) should access to N/P consecutive rows (each one with N/P consecutive columns) from all other $P-1$ processors; and alternatively, each processor has to serve $P-1$ requests from the other processors which should be sequentialised.

3. Write the expression that determines the execution time with P processors, T_P, clearly identifying the contribution of the computation time and the overheads caused by data sharing.

Solution:

Remote accesses before starting the parallel execution in *Region A* are sequentialised because processor 0 can only serve 1 remote access at a time: this is why the cost of a remote access $(t_s + t_w \times N^2/P)$ is multiplied by P-1. Similarly, the P-1 remote accesses to the same processor before starting the parallel execution in *Region B* are also sequentialised because a processor can only serve 1 remote access at a time: this is why the cost of a remote access $(t_s + t_w \times (N/P)^2)$ is multiplied by P-1 too. Therefore, the expression for T_P is:

$T_p = T_P^{comp} + T_P^{data}$

$T_P^{comp} = N^2/P \times t_{body} + N^2/P \times t_{body} = 2 \times N^2/P \times t_{body}$

$T_P^{data} = (P-1) \times (t_s + t_w \times N^2/P) + (P-1) \times (t_s + t_w \times (N/P)^2)$.

Problem 3 (3 points) Assume a multiprocessor system with a hybrid NUMA/UMA architecture. The multiprocessor is composed of 2 identical NUMAnodes, each with 12 Gbytes of main memory. Each NUMAnode has 2 processors, each with its own private cache of 16 Mbytes. Memory and cache lines are 128 bytes wide. Data coherence is maintained using *Write-Invalidate MSI protocol* within each NUMAnode and using a *Write-Invalidate MSU Directory-based* cache coherency protocol among NUMAnodes. **First, we ask you to** answer the following two questions:

1. Compute the total number of bits that are necessary **in each cache memory** to maintain the coherence between caches **inside a NUMAnode**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSI) for each line of cache memory. Each cache memory has $(16 \times 2^{20}) \div 128$ lines, that is 2^{17} lines; therefore the number of bits per cache is $2^{17} \times 2 = 2^{18}$ bits.

2. Compute the total number of bits that are necessary **in each NUMAnode's directory** to maintain the coherence **among NUMAnodes**. Indicate also the function of those bits.

   Solution:

   We need 2 state bits (MSU) and 2 presence bits for each line of main memory. For the overall 12 GB, this is $(12 \times 2^{30}) \div 128$ lines, that is 12×2^{23} lines; therefore the number of bits in the directory is $12 \times 2^{23} \times (2+2) = 3 \times 2^{27}$ bits.

Now, given the following declaration for vector `x`:

```
#define N 1024
int x[N];
```

and assuming that: 1) the initial memory address of vector `x` is aligned to the start of a memory/cache line; 2) the size of an `int` data type is 4 bytes; and 3) processors 0 and 1 belong to NUMAnode0 and processors 2 and 3 belong to NUMAnode1. **We ask you to:**

3. Complete the table in the provided answer sheet with the necessary missing information: type of memory access (`read/write`), affected cache line (numbered from the first position where vector `x` is allocated), access in cache (hit/miss), CPU command for processor k ($PrRd_k/PrWr_k$), Bus transaction(s) from Snoopy in processor k ($BusRd_k/BusRdX_k/BusUpgr_k/Flush_k/Nothing$), cache line states (`I/S/M`), NUMA commands (yes/no), directory entry state (`U/S/M`) and presence bits (0/1, where the lowest ordered bit, the rightmost one, corresponds to NUMAnode0), to keep cache coherence, **AFTER the execution of each** memory access. **Note**: We are not asking for the coherence commands exchanged between NUMAnodes, we are only asking the Bus transactions within NUMAnodes to keep coherence resulted from local or remote memory access to NUMAnodes.

Solution:

Observe that cache line size is 128 bytes and each integer is 4 bytes long. Therefore, 32 consecutive elements of vector x fit in one cache line. In the table below this means that `x[4]`, `x[16]` and `x[20]` belong to the same line (line 0) and `x[32]` is in a different line (line 1).

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state				NUMA commands	Directory entry	
					0	1	2	3		State	Presence bits
Processor 1 writes x[4]	0	Miss	PrWr1	BusRdX1	I	M	I	I	No	M	01
Processor 2 reads x[16]	0	Miss	PrRd2	BusRd2 / Flush1	I	S	S	I	Yes	S	11
Processor 3 writes x[32]	1	Hit	PrWr3	BusUpgr3	I	I	I	M	Yes	M	10
Processor 0 writes x[32]	1	Miss	PrWr0	BusRdX0 / Flush3	M	I	I	I	Yes	M	01
Processor 2 writes x[20]	0	Hit	PrWr2	BusUpgr2	I	I	M	I	Yes	M	10

Problem 4 (2.5 points) Given the following OpenMP code:

```
#define N          (1<<18)    /*  256*1024 */
#define N_THREADS (1<<4)    /*   16 */

int b[N];
int a[N];

#pragma omp parallel num_threads(N_THREADS)
{
  int thid = omp_get_thread_num();
  int chunk = 2;

  for (int ii = thid * chunk; ii < N; ii+= chunk*N_THREADS) {
    for (int i = ii; i < ii+chunk;  i++) {
           b[i] =  a[i];
    }
  }
}
```

Assume an SMP system with 16 CPUs, each with its own cache memory initially empty. To keep caches coherent the system uses a Snoopy–based write–invalidate MSI coherence protocol. Also assume cache lines of 128 bytes, that `int` size is 4 bytes, first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. Finally, we know that thread i runs on CPU i.

Although the code above is correct, its execution generates a lot of bus coherence transactions due to false sharing and bad exploitation of spatial locality, and as a consequence, high execution overheads. **We ask you:**

1. Indicate which threads are accessing to the first 6 elements of vector `a` and `b`.

   Solution:

   For both vectors `a` and `b`, elements 0 and 1 are accessed by thread 0, elements 2 and 3 are accessed by thread 1, elements 4 and 5 are accessed by thread 3, etc.

2. Assuming all processor caches are empty at the beginning of the code. Which types of coherence commands are generated by the snoopy controllers when the code is executed in parallel? Just indicate the name of the coherence transactions and which accesses to variables provoke them. Would it be possible to count the number of coherence transactions of each type? Reason your answer.

   Solution:

   Note that first element of vectors `a` and `b` are placed at the first position of memory line and rest of variables are stored in registers. This means that first element of each vector is aligned to memory line (cache line). However, vector a and b are placed in different memory addresses and then, access to one vector doesn't provoke coherence commands affecting the other vector.

Therefore, accesses to vector b provoke BusRdX and Flush transactions[1]. Accesses to vector a only provoke BusRd transactions. For BusRdX and Flush transactions we cannot exactly count the number of bus transactions that will occur since we don't exactly know the order that each thread will access to the elements of vector b. For BusRd transactions we can say that there will be one per chunk of elements accessed ($N/2$ in total).

Details not needed in your answer to the question: In the case of the BusRdx and Flush transactions we can analyze it a little bit more:

- *For BusRdx:*
  - *the minimum number of transactions is one per chunk of elements accessed ($N/2$ in total).*
  - *the maximum number of transactions is the total number of accesses to vector b (N).*
- *For Flush: The number of Flush transactions will be smaller than the number of BusRdX transactions. Note that the first thread writing for the first time to a memory line does not provoke Flush transaction because there is not a dirty copy of this memory line in the rest of the caches. Later, when there exists a previous dirty copy of the memory line, any write to this memory line that provokes a new BusRdX will also provoke a Flush transaction.*

3. Briefly describe the reason why the execution of the code above leads to a a false sharing situation and a lack of spatial locality exploitation. Propose a modification in the code that avoids both situations at once.

   Solution:

   Cache line size is 128 bytes. Each integer is 4 bytes long. Therefore, every 32 consecutive elements of vector b can fit in one cache line (similarly for vector a). In the case of the code, 16 threads, running in different CPUs, access to consecutive elements (in this case each thread access 2 of the 32 elements that fit in a cache line) of vector b (similarly for vector a).

   - In the case of vector b this may provoke a false sharing situation if two or more threads alternatively write to the same cache line.
   - Lack of locality happens because each thread has a cache miss when accessing the first element of each `chunk`, and only can exploit spatial locality for the second element of the chunk.

   To solve both problems, we only need to set the value of `chunk` to 32. This provokes that threads access to different cache lines and fully exploit spatial locality.

[1] BusUpgr transactions are not possible because we are only writing to vector b

Student name: .....

Answer sheet for **Question 3.3**.

Memory access	Affected line	Hit/Miss	CPU command	Bus transaction(s)	Cache line state				NUMA commands	Directory entry	
					0	1	2	3		State	Presence bits
Processor 1 ..... x[4]	..	Miss	.....	.....	I	...	I	I	No	M	....
Processor 2 ..... x[16]	..	....	.....	.....	...	...	S	...	..	..	....
Processor 3 ..... x[32]	..	Hit	..... .	$BusUpgr_3$	I	I	I	...	Yes	M	....
Processor 0 ..... x[32]	..	....	.....	.....	M	...	...	...	..	..	....
Processor 2 writes x[20]	..	....	.....	.....	...	...	...	...	..	..	....