Part II

Final Exams

PAR – Final Exam – Course 2018/19-Q2

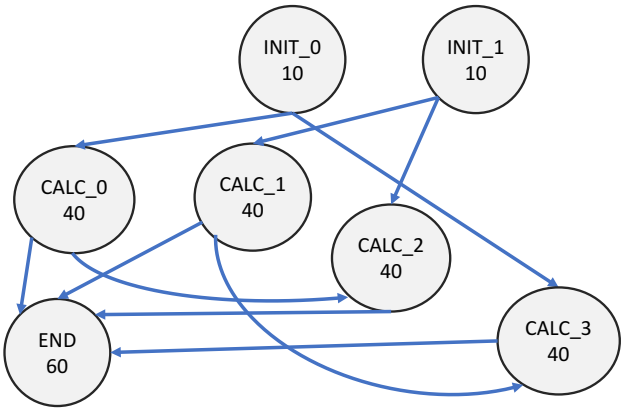
June 18^{th}, 2019

Problem 1 (3 points) Given the following sequential code with calls to *Tareador* to define tasks:

```
#define N 4
int m[N][N];
char taskname[8];

for (int i = 0; i < N/2; i++) {
     sprintf(taskname, "INIT_%d", i);
     tareador_start_task (taskname);
     for (int k = 0; k < N; k++) {
         m[i][k] = foo(i, k); // no access to m inside function foo
         m[N-i-1][k] = m[i][k];
     }
     tareador_end_task(taskname);
}

for (int i = 0; i < N; i++) {
    sprintf(taskname, "CALC_%d", i);
    tareador_start_task(taskname);
    for (int k = 0; k < N; k++) {
        if (i < N/2)
            m[i][k] += calculate_something (m[i][k]);
        else
            m[i][k] += calculate_something (m[i-N/2][k]);
    }
    tareador_end_task(taskname);
 }

 tareador_start_task("END");
 save_results(m);
 tareador_end_task("END");
```

Assume: 1) the execution of functions `calculate_something` and `save_results` do not modify the input parameters; and 2) the execution of each `INIT_i` task takes 10 time units, of each `CALC_i` task takes 40 time units and of the `END` task takes 60 time units. **We ask you:**

1. (1 point) Draw the complete *Task Dependence Graph* (TDG) using the task identifiers dynamically set in the code.
   Solution:



2. (1 point) Compute the values for T_1, T_∞, potential *Parallelism* and P_{min}.
   Solution:
   $T_1 = 2 \times 10 + 4 \times 40 + 60 = 240$
   critical path = INIT_0, CALC_0, CALC_2, END
   $T_\infty = 10 + 2 \times 40 + 60 = 150$
   $Parallelism = T_1 \div T_\infty = 240/150 = 1.6$
   $P_{min} = 2$

Now assume that 1) memory of the parallel system is physically distributed so that the access to data in different processors introduces a data–sharing overhead; this overhead follows the model explained in class: $t_{comm} = t_s + B \times t_w$ being t_s and t_w startup and transfer time, respectively, and B the number of elements accessed. And 2) a *first touch* allocation policy at page level is used by the operating system to initially place data in distributed memory. **We ask you:**

3. (1 point) Define the assignment of tasks to the P_{min} processors calculated before, minimizing the data–sharing overhead, and find the expression for the execution time T_p for $p = P_{min}$ taking into account this overhead.
   Solution:
   Task allocation to P_{min} processors:
   $P_0 = INIT\_0, CALC\_0, CALC\_2, END$
   $P_1 = INIT\_1, CALC\_1, CALC\_3$
   $T_p^{calc} = 10 + 40 + 40 + 60 = 150$
   The critical path is in P_0 and the remote data access from P_0:
   task CALC_2 access remotely row i = 2 allocated in P_1
   task END access locally rows i = 0 and i = 2 and has to ask to P_1 row i = 1 and latest version of row i = 3 , so the expression for the communication time is:
   $T_p^{comm} = (t_s + 4 \times t_w) + (t_s + 8 \times t_w)$
   Finally the expression for the execution time on two processors:
   $T_2 = 150 + 2 \times t_s + 12 \times t_w$

Problem 2 (4 points) In this problem you have to parallelize function `iter_distribute`. This function copies vector S into vector D in such a way that all those elements $S[i]$ with the same value for $S[i]\%256$ are stored in consecutive positions of D. Therefore, at the end of the function, there will be in D all the elements of S organized in 256 groups of elements: first all those with value %256 equal to 0, then those with value %256 equal to 1, ... up to those with value %256 equal to 255. In order to do this copy, the implementation provides a vector C which is initialized in function `preprocessing`; each element $C[value]$ indicates the initial position in D to store all those elements $S[i]$ whose $S[i]\%256 = value$.

```
#define N 1024*1024*1024
unsigned int S[N], D[N], C[256];

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
    unsigned int i, value;

    for (i=0; i<n; i++) {
        value = S[i]%256;
        D[C[value]] = S[i];
        C[value]++;
    }
}

void main() {
    ...
    preprocessing(S, C, N); // initialization of S and C
    ...
    iter_distribute(S, N, C, D);
    ...
}
```

Assuming that it is not important the order of the elements inside the same group in D (i.e. the elements of S can be written in different relative order in D in the parallel program than the order in the sequential code), **we ask you:**

1. (1 point) Write an OpenMP parallel implementation of function `iter_distribute` that follows a *Geometic Cyclic Data Decomposition* of the Input vector `S`. You should maximize the parallelism that your solution offers minimizing the serialization that is introduced by synchronization, if any.
   Solution:

```
#include <omp.h>
#define N 1024*1024*1024

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 omp_lock_t locks_C[256];

 for(i=0;i<256;i++)
   omp_init_lock(&locks_C[i]);

 #pragma omp parallel private(value, i)
 {
   unsigned int myid = omp_get_thread_num();
   unsigned int nt   = omp_get_num_threads();

   for (i=myid; i<n; i+=nt) {
     value = S[i]%256;

     omp_set_lock(&locks_C[value]);
     D[C[value]] = S[i];
     C[value]++;
     omp_unset_lock(&locks_C[value]);
   }
 }

 for(i=0;i<256;i++)
   omp_destroy_lock(&locks_C[i]);

}
```

2. (1 point) Write an alternative OpenMP parallel implementation of function `iter_distribute` that follows a *Geometric Block Data Decomposition* of the Output vector `C`. Consequently, each thread uses and updates only a part of C, and then, will only write to a part of D, depending of the part of C the thread works with. Your parallel implementation has to ensure a proper load balance of the data distribution (i.e. no more than 1 element of difference) and maximize the parallelism that your solution offers minimizing the serialization that is introduced by synchronization, if any.
   Solution:

```
#include <omp.h>
#define N 1024*1024*1024

void iter_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
 unsigned int i, value;

 #pragma omp parallel private(value, i)
 {
   unsigned int myid = omp_get_thread_num();
   unsigned int nt   = omp_get_num_threads();
   unsigned int C_nt= 256/nt;
   unsigned int rest_nt = 256%nt;
   unsigned int C_start = myid*C_nt + ((rest_nt>myid)?myid:rest_nt);
```

```
   unsigned int C_end   = C_start + C_nt + (myid<rest_nt);

   for (i=0; i<n; i++) {
     value = S[i]%256;
     if ((value >= C_start) && (value < C_end))
     {
        D[C[value]] = S[i];
        C[value]++;
     }
   }
 }

}
```

3. (2 points) Finally, we have created a recursive divide–and–conquer sequential version of previous code. Write an OpenMP parallel code for function `rec_distribute` and main program adding the necessary code and directives to implement a *recursive tree task decomposition*. Your parallel code should include a cut–off mechanism based on recursion depth, allowing parallel recursive calls for depths smaller than `MAX_DEPTH`. Your parallel code should minimize the serialization that is introduced by synchronization, if any.

```
#define N 1024*1024*1024
unsigned int S[N], D[N], C[256];

void rec_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D) {
    unsigned int i, value;
    unsigned int n2 = n/2;
    if (n==1) {
        value = S[0]%256;
        D[C[value]] = S[0];
        C[value]++;
    } else {
        rec_distribute(S, n2, C, D);
        rec_distribute(&S[n2], n-n2, C, D);
    }
}


void main() {
    ...
    preprocessing(S, C, N); // initialization of S and C
    ...
    rec_distribute(S, N, C, D);
    ...
}
```

Solution:

```
#include <omp.h>
#define N 1024*1024*1024
omp_lock_t locks_C[256];

// ... As above
void rec_distribute(unsigned int *S, int n, unsigned int C[256], unsigned int *D,
                    unsigned int depth) {
 unsigned int i, value;
 unsigned int n2 = n/2;
 if (n==1) {
   value = S[0]%256;
      omp_set_lock(&locks_C[value]);
      D[C[value]] = S[0];
      C[value]++;
```

```
      omp_unset_lock(&locks_C[value]);
 } else {
    if (!omp_in_final()) {
        #pragma omp task final(depth>=MAX_DEPTH)
        rec_distribute( S, n2, C, D, depth+1);
        #pragma omp task final(depth>=MAX_DEPTH)
        rec_distribute( &S[n2], n-n2, C, D, depth+1);
    } else {
        rec_distribute( S, n2, C, D, depth+1);
        rec_distribute( &S[n2], n-n2, C, D, depth+1);
    }
 }
}

unsigned int S[N], D[N], C[256];

void main() {
    ...
    preprocessing(S, C, N); // initialization of S and C
    ....
    for(i=0;i<256;i++) omp_init_lock(&locks_C[i]);

    #pragma omp parallel
    #pragma omp single
    rec_distribute(S, N, C, D, 0);

    for(i=0;i<256;i++) omp_destroy_lock(&locks_C[i]);
    ...
}
```

Problem 3 (3 points) Given a NUMA muliprocessor system with X NUMA nodes, each NUMA node including Y sockets sharing the access to the node's main memory, each socket with Z cores sharing the access to a per-socket cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing write–invalidate MSI. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing write–invalidate MSU. The following table summarises the main characteristics of the system, including the total number of bits devoted to keep coherence at the two levels (inside and among NUMA nodes).

System characteristic	**Size**
Number of NUMA nodes	X nodes
Size of main memory per NUMA node	8 GB (gigabytes)
Number of bits per NUMA node devoted to coherence in the directory (no data)	2^{32} bits
Number of sockets per NUMA node	Y sockets
Size of each per-socket cache memory	4 MB (megabytes)
Number of bits per NUMA node devoted to coherence among sockets (no data)	2^{20} bits
Number of cores per socket	Z cores
Cache and memory line size	32 B (bytes)

Question 3.1 (1 point) We ask you to compute the number of NUMA nodes X composing the system and the number of sockets Y per NUMA node. With the information provided, is it possible to determine the number of cores Z per socket? In case of affirmative answer, please compute that number.

Solution: In each NUMA node the memory is able to store $8GB \div 32$ memory lines. This is $2^3 \times 2^{30} \div 2^5 = 2^{28}$ lines. For each line in memory the directory needs to store 2 bits to keep the state of the line (MSU) and X presence (sharers) bits (one per NUMA node). Since the total number of bits per NUMA node devoted to the directory is 2^{32} bits, we can formulate $2^{28} lines/node \times (2 + X) bits/line = 2^{32}$ bits; so $X = 14$.

Similarly, each cache memory inside a NUMA node is able to store $4MB \div 32$ memory lines. This is $2^2 \times 2^{20} \div 2^5 = 2^{17}$ lines. For each line in the cache the snoopy needs to store 2 bits to keep the state of the line (MSI). Since the total number of bits per NUMA node devoted to the snoopy coherence is 2^{20} bits, we can formulate $Y sockets/node \times 2^{17} lines/socket \times 2 bits/line = 2^{20}$ bits; so $Y = 4$.

Finally, there is not sufficient information to conclude the number of cores per socket, since they are all sharing the access to the same cache.

Assume that two different cores in the previous system, one in $NUMAnode_i$ and another in $NUMAnode_j$ are trying to atomically update variable `sum` with the following code:

```
#pragma omp atomic
sum += local_result;
```

Also assume that the home NUMA node for variable `sum` is $NUMAnode_k$ and that initially there are no copies of the memory line containing variable `sum` in any cache memory.

Question 3.2 (1 point) Write an alternative code in C (with no OpenMP pragmas) to implement the atomic access that is done during the execution of previous code using the `atomic` pragma in OpenMP, making use of the following low–level *load-linked store-conditional* synchronisation primitives:

```
int load_linked (int *address);
int store_conditional (int *address, int value); // returns 1 in case of success
```

Solution: We simply need to chain the two low–level synchronization constructs and check whenever the store is successfully completed, as follow:

```
int ret;
do {
        int value = load_linked (&sum);
        value += local_value;
        ret = store_conditional (&sum, value);
} while (ret == 0);
```

Question 3.3 (1 point) Fill in the table in the solutions page with the sequence of coherence actions that will take place, both within and across NUMA nodes, if the two atomic updates by the core in $NUMAnode_i$ and by the core in $NUMAnode_j$ are **not overlapped in time**, so that first the core in $NUMAnode_i$ atomically updates variable `sum` followed by the core in $NUMAnode_j$ atomically updating variable `sum`. Clearly indicate, for each instruction, the order in which the coherence actions occur (e.g. `1-BusRd; 2-RdReq(k->j); 3-Dreply(j->k)`), how the state of the line in the directory will change and how the state of the lines holding the different copies in cache will change. For the execution of `load_linked` and `store_conditional` the processor issues a `PrRd` and `PrWr` command, respectively.

Time	NUMA Node i				NUMA Node j				NUMA node k					
	Instruction	Socket cache state	Bus transaction	NUMA transaction	Instruction	Socket cache state	Bus transaction	NUMA transaction	NUMA transaction	Socket cache state	Directory state	Sharers list		
		sum				sum				sum	sum	k	j	i
0		-				-				-	U	0	0	0
1	**load_linked**	S	1-BusRd	2-RdReq(i->k)					3-Dreply(k->i)		S	0	0	1
2	**store_conditional**	M	1-BusUpgr	2-UpgrReq(i->k)					3-Ack(k->i)		M	0	0	1
3		S	4-BusRd, 5-Flush	6-Dreply(i->k)	**load_linked**	S	1-BusRd	2-RdReq(j->k)	3-Fetch(k->i), 7-Dreply(k->j)		S	0	1	1
4		I	4-BusUpgr	5-Ack(i->k)	**store_conditional**	M	1-BusUpgr	2-UpgrReq(j->k)	3-Invalidate(k->i) 6-Ack(k->j)		M	0	1	0

Solution sheet for Problem 3

SURNAME:

NAME:

Time	NUMA Node i				NUMA Node j				NUMA node k					
	Instruction	Socket cache state	Bus transaction	NUMA transaction	Instruction	Socket cache state	Bus transaction	NUMA transaction	NUMA transaction	Socket cache state	Directory state	Sharers list		
		sum				sum				sum	sum	k	j	i
0		-				-				-	U	0	0	0
1	**load_linked**													
2	**store_conditional**													
3					**load_linked**									
4					**store_conditional**									

(1) Cache line state: M (modified), S (shared) or I (invalid)
(2) Coherence commands inside NUMA node: BusRd, BusRdX, BusUpgr and Flush
(3): Line state in node memory: M (modified), S (shared) or U (uncached)
(4) Coherence commands between NUMA nodes a and b: RdReq(a->b), WrReq(a->b), UpgrReq(a->b), Dreply(a->b), Ack(a->b), Fetch(a->b) and Invalidate(a->b)

PAR – Final Exam – Course 2019/20-Q1
January 16^{th}, 2020

Problem 1 (2 points)

The following code excerpt instrumented with *Tareador* belongs to a program we want to parallelize. It simulates the heat diffusion equation using the *Gauss-Seidel* method. We have developed a blocked implementation that creates $N \times M$ blocks:

```
#define N ...
#define M ...
#define lowerb(id, p, n)  ( id * (n/p) + (id < (n%p) ? id : n%p) )
#define upperb(id, p, n)  ( lowerb(id, p, n) + (n/p) + (id < (n%p)) - 1 )

/* Blocked Gauss-Seidel solver: Compute 1 subblock */
double compute_GS_block( double *u, unsigned sizex, unsigned sizey,
                         int i_lb, int i_ub, int j_lb, int j_ub) {
  double unew, diff, blocktotal=0.0;
  for (int i=max(1, i_lb); i<= min(sizex-2, i_ub); i++)
    for (int j=max(1, j_lb); j<= min(sizey-2, j_ub); j++) {
      unew= 0.25 * ( u[     i*sizey     + (j-1) ]+  // left
                     u[     i*sizey     + (j+1) ]+  // right
                     u[ (i-1)*sizey     + j     ]+  // top
                     u[ (i+1)*sizey     + j     ]); // bottom
      diff = unew - u[i*sizey+ j];
      blocktotal += diff * diff;
      u[i*sizey+j]=unew;
    }
  return(blocktotal);
}

/* Blocked Gauss-Seidel solver: one iteration step */
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
    double total=0.0;
    int howmanyX = N;
    int howmanyY= M;

    tareador_disable_object(&total);
    for (int blockidX = 0; blockidX < howmanyX; ++blockidX)
      for (int blockidY = 0; blockidY < howmanyY; ++blockidY) {
        int i_lb = lowerb(blockidX, howmanyX, sizex);
        int i_ub = upperb(blockidX, howmanyX, sizex);
        int j_lb = lowerb(blockidY, howmanyY, sizey);
        int j_ub = upperb(blockidY, howmanyY, sizey);
        tareador_start_task("GS Block");
        total+=compute_GS_block( u, sizex, sizey, i_lb, i_ub, j_lb, j_ub);
        tareador_end_task("GS Block");
      }
    tareador_enable_object(&total);
    return total;
}

int main() {
    init();
    tareador_ON();
    relax_gauss(...);
    tareador_OFF();
 }
```

We ask you to answer the following questions:

1. (0.5 points) Draw the Task Dependence Graph (TDG) that would be generated by *Tareador* for 1 invocation of routine `relax_gauss` considering $N = 3$ and $M = 5$ given the implementation shown above. Note: You can ommit the labels with details provided by *Tareador* within each node of the TDG.

Solution:

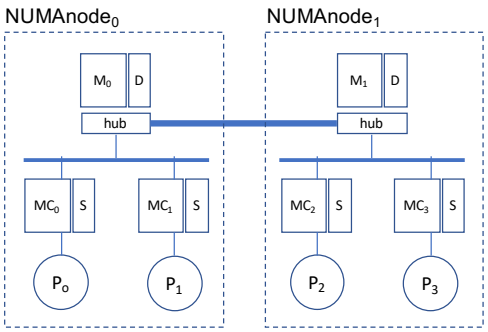
2. (0.5 points) In view of the TDG above, give a general expression for P_{min} as a function of N and M. Hint: Consider how the TDG would look like if N was larger than M.

   Solution: To schedule the tasks in the critical path without delays we need as many processors as the smallest of N and M. Thus, $P_{min} = min(N, M)$.

3. (1 point) Give an expression for the parallel execution time as a function of N and M considering that: 1) the execution time of each invocation of routine `compute_GS_block` and update of variable `total` takes T_c time units; 2) there exists a cost for syncronization between tasks so that the overhead that a task has to pay to get notified that ALL its predecessor tasks have finished is T_{sync}; 3) the cost for task creation is negligible; 4) P_{min} processors are used.

   Solution: $T_{P_{min}} = (N + M - 1) \times T_c + (N + M - 2) \times T_{sync}$

Problem 2 (2 points) Consider a parallel architecture composed of two NUMA nodes, each with its own main memory, directory to keep coherence between NUMA nodes (write-invalidate MSU) and two processors with their own cache memory and snoopy to keep coherence within each node (write-invalidate MSI). For the purposes of this problem, you can assume infinite sizes for both main and cache memories.



Legend:
NUMAnode$_i$: NUMA node i with two processors
M$_i$: main memory for NUMAnode$_i$
D: directory associated to M
hub: interconnect between NUMA nodes
P$_j$: processor j inside NUMA node
MC$_j$: cache memory local to processor P$_j$
S: snoopy associated to MC

Coherence commands:
- Snoopy: BusRd(j), BusRdX(j), BusUpgr(j) and Flush(j), being j the snoopy/cache number doing the action or hub
- Hub/directoty: RdReq(i→j), WrReq(i→j), UpgrReq(i→j), Dreply(i→j), Ack(i→j), Fetch(i→j), Invalidate(i→j) and WriteBack(i→j), from NUMAnode$_i$ to NUMAnode$_j$

Within each invocation of routine `relax_gauss` in the previous problem there is an update of a shared variable named `total`. Being aware of the existence of two NUMA nodes we have applyied low-level synchronizations using the `load_linked (ll)` and the `store_conditional (sc)` primitives seen during the course:

```
double load_linked (double *address);
int store_conditional (double *address, double value); // returns 0 if fails; 1 otherwise
```

```
  int ret;
  double local_contribution, total=0.0;
  ...
  local_contribution = compute_GS_block( u, sizex, sizey, i_lb, i_ub, j_lb, j_ub);
  do {
       double value = load_linked (&total);
```

```
      value += local_contribution;
      ret = store_conditional (&total, value);
  } while (ret == 0);
  ...
```

Let us assume that: 1) the *home node* for variable `total` is NUMAnode$_0$; 2) the sequence of synchronization operations starts with the `ll` and the `sc` in core 3 within NUMAnode$_1$ and is later followed by the `ll` and the `sc` in core 1 within NUMAnode$_0$; 3) for the execution of `load_linked` and `store_conditional` the processor issues a `PrRd` and `PrWr` command, respectively.

We ask you to fill in the table provided in the answer sheet with the sequence of coherence actions that will take place, both within and across NUMA nodes. Indicate, for each instruction, the order in which the coherence actions occur and the processor or hub number performing the action (e.g. `1-BusRd(2); 2-RdReq(1->0); 3-Dreply(0->1); 4-...`), and the resulting state of the line in the directory as well as in any cache lines involved in storing the copies.

Solution:

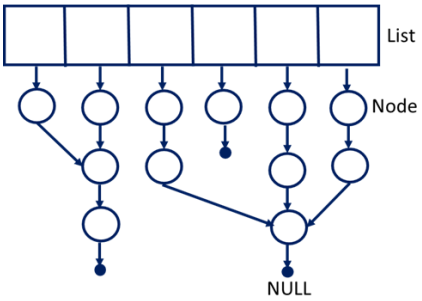
Time	NUMA Node 1					NUMA Node 0							
	Instruction	MC3 line state	MC2 line state	Bus transactions	NUMA transactions	Instruction	MC1 line state	MC0 line state	Bus transactions	NUMA transactions	Directory state	Sharers list	
		total	total				total	total			total	1	0
0		-	-				-	-			U	0	0
1	**load_linked (in core 3)**	S		1-BusRd(3)	2-RdReq(1→0)					3-Dreply(0→1)	S	1	0
2	**store_conditional (in core 3)**	M		1-BusUpgr(3)	2-UpgrReq(1→0)					3-Ack(0→1)	M	1	0
3		S		3-BusRd(hub) 4-Flush(3)	5-Dreply(1→0)	**load_linked (in core 1)**	S		1-BusRd(1)	2-Fetch(0→1)	S	1	1
4		I		3-BusUpgr(hub)	4-Ack(1→0)	**store_conditional (in core 1)**	M		1-BusUpgr(1)	2-Invalidate(0→1)	M	0	1

Problem 3 (3 points) Given the following C code that calculates the sum of all the elements in a structure of type List:

```
#define  N  ...

typedef struct Node {
    float value;
    int footprint; // initialized to value 0
    struct Node *next;
} List;

float compute_sequential (struct Node *p) {
     float sum = 0.0;
     int end = 0;
     while (p != NULL && end == 0) {
          int x = ++p->footprint;
          if (x == 1)  // to ensure value is accumulated only once
               sum = sum + heavy_calculation(p->value);
          else
               end = 1;
          p = p-> next;
     }
     return sum;
}

float process_vector (List *v[N]) {
     float res = 0.0;
     for (int i = 0; i < N; i++)
         res = res + compute_sequential (v[i]);
     return res;
}
```

```
void main() {
    List *v[N];
    // initialize footprint to value 0
    ...
    float total = process_vector (v);
    ...
}
```

where type `List` represents a vector of lists with the particularity that some lists are connected (two nodes from different lists can have the same following node in the list) as shown in the figure below:



We ask you to:

1. Write an OpenMP parallel version of the `process_vector` function using an iterative task decomposition strategy trying to maximize load balancing.

   Solution:

   ```
   float process_vector (List *v[N]) {
        float res = 0.0;
        #pragma omp parallel
        #pragma omp single
        #pragma omp taskloop grainsize(1) reduction (+:res)
        for (int i = 0; i < N; i++)
            res = res + compute_sequential (v[i]);
        return res;
   }
   ```

   In order to be correct, the update to footprint field in compute_sequential has to be done atomically:

   ```
   float compute_sequential (struct Node *p) {
        float sum = 0.0;
        int end = 0;
        while (p != NULL && end == 0) {
             #pragma omp atomic
             int x = ++p->footprint;
             if (x == 1)  // to ensure value is accumulated only once
                  sum = sum + heavy_calculation(p->value);
             else
                  end = 1;
             p = p-> next;
        }
        return sum;
   }
   ```

2. Write an OpenMP parallel version of `process_vector_rec` function following a *divide and conquer* task decomposition strategy and using the following sequential recursive version of `process_vector`:

   ```
   float process_vector_rec (List *v[N], int n) {
        float res = 0.0;
   ```

```
     if (n <= MIN_SIZE)
         for (int i = 0; i < n; i++)
             res = res + compute_sequential (v[i]);
     else {
         int n2 = n / 2;
         float res1 = process_vector_rec (v, n2);
         float res2 = process_vector_rec (v+n2, n-n2);
         res = res1 + res2;
     }
     return res;
}

int main() {
    List *v[N];
    ...
    float total = process_vector_rec (v, N);
    ...
}
```

Solution:

```
float process_vector_rec (List *v[N], int n) {
     float res = 0.0;
     if (n <= MIN_SIZE)
         for (int i = 0; i < n; i++)
             res = res + compute_sequential (v[i]);
     else {
         int n2 = n / 2;
         #pragma omp task shared(res1)
         float res1 = process_vector_rec (v, n2);
         #pragma omp task shared(res2)
         float res2 = process_vector_rec (v+n2, n-n2);
         #pragma omp taskwait
         res = res1 + res2;
     }
     return res;
}
```

```
int main() {
    List *v[N];
    ...
    #pragma omp parallel
    #pragma omp single
    float total = process_vector_rec (v, N);
    ...
}
```

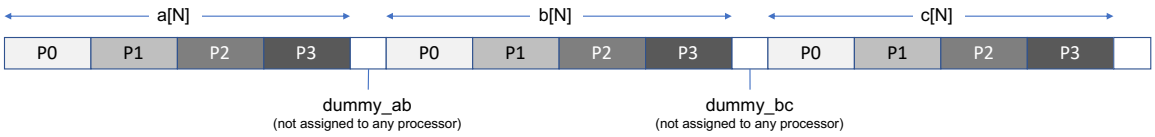
Again, in order to be correct, the update to footprint field in compute_sequential has to be done atomically as shown in the previous part.

Problem 4 (3 points) Assume the following incomplete version for an OpenMP parallel program:

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct SoA {
     int a[N];
     int dummy_ab[...];  // to complete
     int b[N];
     int dummy_bc[...];  // to complete
```

```
     int c[N];
};
struct SoA dataElements;
...
#pragma omp parallel
{
     int lower = ...;  // to complete
     int upper = ...;  // to complete
     for (int i = lower, i < upper; i++)
          dataElements.c[i] = foo(i, dataElements.a[i], dataElements.b[i]);
}
...
```

1. We ask you to complete the code above if we want to apply a *geometric block data decomposition* to each vector `a`, `b` and `c`, as shown in the figure below for the case of 4 processors:



   Your parallel code should make sure that vectors are properly aligned in memory (i.e. each one starts at the beginning of a cache line) in order to guarantee the proposed data decomposition; you can assume that vector `a` is already aligned in memory. To simplify the problem, you can also assume that the number of processors will always divide `N` perfectly. In particular you need to complete the lines indicated and add extra code, if necessary. As indicated in the code, each integer element occupies 4 bytes of memory and a cache line is 64 bytes long.

   Solution: Each one of the vectors has $N = 1000$ elements and occupies $N \times INT\_SIZE$ bytes, that is 4000 bytes. This number of bytes fits in $4000 \div CACHE\_LINE = 62,5$ cache lines. To ensure that the next vector starts in a new cache line, we should add a padding of half a line, that is 32 bytes, or equivalently 8 integer elements in each padding vector.

   Since the number of processors always divides N perfectly, we don't need to add code to balance the number of elements assigned to each thread.

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct SoA {
     int a[N];
     int dummy_ab[8]; // (CACHE_LINE - (N*INT_SIZE)%CACHE_LINE) / INT_SIZE
     int b[N];
     int dummy_bc[8];
     int c[N];
};
struct SoA dataElements;
...
#pragma omp parallel
{
     int myid = omp_get_thread_num();
     int howmany = omp_get_num_threads();
     int BS = N / howmany;
     int lower = myid * BS;
     int upper = lower + BS; // no need to add code to balance the number of iterations
     for (int i = lower, i < upper; i++)
          dataElements.c[i] = foo(i, dataElements.a[i], dataElements.b[i]);
}
...
```

In order to avoid the need of introducing padding, the developer decided to change the definition of `dataElements` from *structure of arrays* to *array of structures*, as follows:

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct AoS {
     int a;
     int b;
     int c;
};
struct AoS dataElements[N];
...
#pragma omp parallel
{
     int lower = ...;  // to complete
     int upper = ...;  // to complete
     int step = ...;   // to complete
     for (int ii = lower; ii < upper; ii += ...)   // to complete
          for (int i = ii; ...; ...)               // to complete
               dataElements[i].c = foo(i, dataElements[i].a, dataElements[i].b);
}
...
```

In addition, the developer detected that function `foo` was introducing a monotonically increasing load unbalance in the computation (i.e. the computation time increases with the value of variable `i`), so he/she proposed to try a *geometric block-cyclic data decomposition* applied to vector `dataElements`. We ask you (the following two questions are independent, you can answer the third one assuming a generic answer from the second one):

2. Decide the number of consecutive elements of vector `dataElements` assigned to each processor in the *geometric block-cyclic data decomposition* that avoids false sharing and reduces load unbalance.

   Solution: Each element of vector `dataElements` occupies 12 bytes (3 integer elements). To avoid false sharing we should assign to each processor a number of consecutive elements that fit in a number of complete cache lines. In this case this is $mcm(12, 64) = 192$ (minimum common multiple), which corresponds to 16 elements. Other computations that also give as result 16, as for example the number of integers in a cache line ($64/4 = 16$), have not been considered correct answers.

3. Complete the code above in order to implement a *geometric block-cyclic data decomposition*.

   Solution:

```
#define N 1000
#define CACHE_LINE 64
#define INT_SIZE 4

struct AoS {
     int a;
     int b;
     int c;
};
struct AoS dataElements[N];
...
#pragma omp parallel
{
     int myid = omp_get_thread_num();
     int howmany = omp_get_num_threads();
     int BS = 16;  // result from previous question
     int lower = myid * BS;
     int upper = N;
     int step = howmany * BS;
     for (int ii = lower; ii < upper; ii += step)
          for (int i = ii; i < min(N, ii+BS); i++)
               dataElements[i].c = foo(i, dataElements[i].a, dataElements[i].b);
}
...
```

Solution sheet for Problem 2

SURNAME:

NAME:

Time	NUMA Node 1					NUMA Node 0							
	Instruction	MC3 line state	MC2 line state	Bus transactions	NUMA transactions	Instruction	MC1 line state	MC0 line state	Bus transactions	NUMA transactions	Directory state	Sharers list	
		total	total				total	total			total	1	0
0		-	-				-	-			U	0	0
1	**load_linked (in core 3)**												
2	**store_conditional (in core 3)**												
3						**load_linked (in core 1)**							
4						**store_conditional (in core 1)**							

(1) Cache line state: M (modified), S (shared) or I (invalid)
(2) Coherence commands inside NUMA node: BusRd, BusRdX, BusUpgr and Flush
(3): Line state in node memory: M (modified), S (shared) or U (uncached)
(4) Coherence commands between NUMA nodes i and j: RdReq(i→j), WrReq(i→j), UpgrReq(i→j), Dreply(i→j), Ack(i→j), Fetch(i→j) and Invalidate(i→j)

PAR – Final Exam: Part 1 – Course 2020/21-Q1
January 18^{th}, 2021

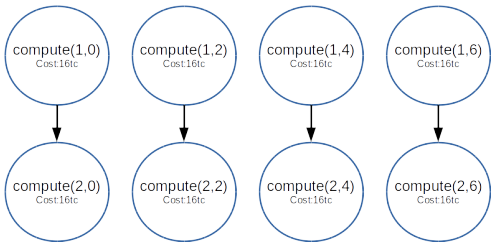
Problem 1 (5 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#define MAX_ROWS 4
#define MAX_VALUE 8
#define BS 2
   ...
   // compute loops
   for (i=1; i<MAX_ROWS-1; i++)
      for (jj=0; jj<MAX_VALUE; jj+=BS)
      {
          sprintf(stringMessage,"compute(%d,%d)",i,jj);
          tareador_start_task(stringMessage);
          for (j=jj; j<jj+BS; j++)
             A[i][j] = A[i][j] + 2*A[i-1][j] - 2*A[i+1][j]; // Cost 8*tc
          tareador_end_task(stringMessage);
      }
   ...
```

We ask you to answer the following questions:

1. Draw the Task Dependence Graph (TDG) assuming the value for the constants and the *Tareador* task definition in the program. In the TDG, annotate each node with the name of the corresponding task (`compute(i,jj)`) and its cost.

   Solution: The i loop only performs two iterations, generating 4 tasks (jj loop) in each of them. There are not dependences among `compute(i,jj)` task instances with the same value of i but there are true dependences due to the access to $A[i-1][j]$, between `compute(i,jj)` - `compute(i+1,jj)`, for jj in $\{0, 2, 4, 6\}$.

2. Compute the T_1, T_∞ and P_{min} metrics associated to the TDG obtained in the previous question.

   Solution: The sum of the cost of all the tasks, executed in one only processor, determines T_1. Each task executes two iterations of the loop body (j loop), therefore:

   $$T_1 = 8 \times 2 \times 8 \times t_c = 128 \times t_c$$

   T_∞ is defined by the minimum cost needed to execute all the tasks of the TDG when using infinite resources. In our case, the critical path is composed by any of the pair of tasks `compute(1,jj)` - `compute(2,jj)`, for jj in $\{0, 2, 4, 6\}$.

   $$T_\infty = 2 \times 2 \times 8 \times t_c = 32 \times t_c$$

   For the given TDG, and the T_∞ obtained, we need 4 processors to execute in parallel all `compute(1,jj)` tasks, or all tasks `compute(2,jj)` in such a way we can achieve the T_∞ time.

   $$P_{min} = 4$$

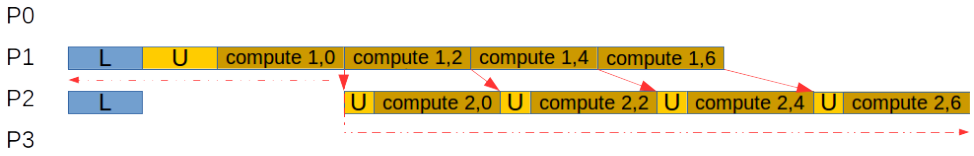
3. Write the expression that determines the execution time T_4 for the program, clearly indicating the contribution of the computation time T_4^{comp} and the data sharing overhead T_4^{mov}, for the following assignment of tasks to threads and processors: tasks `compute(1,jj)` are assigned to `thread 1` (which runs on processor 1) and tasks `compute(2,jj)` are assigned to `thread 2` (which runs on processor 2); threads 0 and 3, mapped to processors 0 and 3, respectively, have no tasks assigned to them.

Tasks	thread	processor
none	0	0
compute(1,0), compute(1,2) compute(1,4), compute(1,6)	1	1
compute(2,0), compute(2,2) compute(2,4), compute(2,6)	2	2
none	3	3

You can assume: 1) a distributed-memory architecture with **4 processors**; 2) matrix A is initially distributed by rows (**row i to processor i**); 3) once the loop is finished, you don't need to return the matrix to their original distribution; 4) data sharing model with $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start-up time and transfer time of one element, respectively; and 5) the execution time for a single iteration of the innermost loop body takes $8 \times t_c$.

Solution: In order to compute the model, we have done the initial communication at the beginning of the execution, as seen at class. Remember, initial communication is not due to true dependences but necessary due to the mapping of the data in a remote processor. However, there are other possibilities that may be more efficient to overlap computation and initial communication; all of them have been considered correct. On the other hand, due to the dependences, `thread 2` has to wait for task `compute(1,jj)` (executed by `thread 1`) to execute task `compute(2,jj)`. Once task `compute(1,jj)` is executed by `thread 1`, `thread 2` will read the upper boundary data of `BS` elements to execute task `compute(2,jj)`.

You can see the initial communication, the executed tasks, the synchronization between tasks with solid red arrow, and the communication during the computation in the following time diagram:



P1 and P2's L : Initial communication: MAX_VALUE-element lower boundary
P1's U: Initial communication - MAX_VALUE-element upper boundary
P2's U: Communication due to dependency - BS-element upper boundary

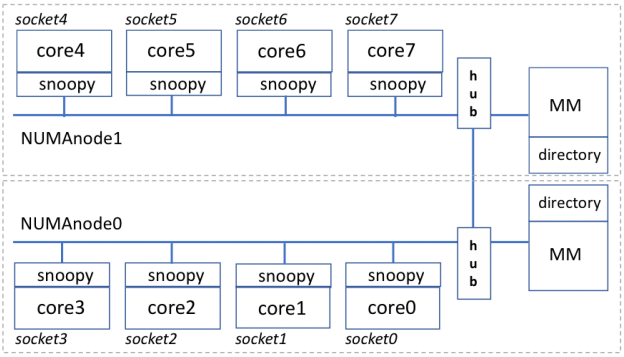
Note that we have added a dashed red line to indicate one possible way to compute the critical path. The critical path cost includes T_4^{comp} and T_4^{mov}:

$T_4^{comp} = 5 \times 2 \times 8 \times t_c$

$T_4^{mov} = 2 \times (t_s + MAX\_VALUE \times t_w) + 4 \times (t_s + BS \times t_w)$

$T_4 = T_4^{comp} + T_4^{mov}$

Problem 2 (5 points) Assume a multiprocessor system composed of two NUMA nodes, each with four sockets and a shared memory (MM) of 8 GB (total MM size is 16 GB). Each socket has one core with a cache memory of 4 MB. The cache line size is 64 bytes. Data coherence within each NUMA node is guaranteed by a Write-Invalidate MSI protocol with a Snoopy attached to each cache memory; data coherence between the two NUMA nodes is guaranteed by a directory-based MSU protocol.



1. Compute the total number of bits that are necessary in each cache memory to maintain the coherence between the caches inside a NUMA node.

   Solution: We need 2 bits for the cache line state. The possible values are: M (Modified), S (Shared) and I (Invalid). The total number of lines in a cache is: cache size / cache line size $= 4 * 2^{20}/2^6 = 2^{16}$, so the total number of bits per cache $= 2 * 2^{16}$.

2. Compute the total number of bits that are necessary in each node directory to maintain the coherence among NUMA nodes.

   Solution: We need 4 bits per directory entry (that corresponds to each memory line: 2 bits for the line state with possible values: M (Modified), S (Shared), U (Uncached) and 2 bits for the data sharing (presence) bits (one bit per NUMA-node). Total number of lines in main memory per NUMA-node is : $8 * 2^{30}/2^6 = 2^{27}$, so the total number of bits per NUMA-node $= 4 * 2^{27}$ bits.

3. Given the following declaration for vector `v`:

   ```
   #define N 64
   int v[N];
   ```

   and assuming that: 1) the initial address of vector `v` is aligned with the start of a cache line; 2) vector `v` is entirely allocated in MM_0 (i.e. the portion of shared memory in **NUMA node 0**); 3) the size of an `int` data type is 4 bytes; and 4) all cache memories are empty at the beginning of the program. **We ask you to** fill in the table in the provided answer sheet with the sequence of processor commands (column *Core*), bus transactions within NUMA nodes (column *Snoopy*), *Yes* or *No* in column *Directory* to indicate if there are transactions between NUMA nodes, the presence bits and state in the directory entry associated to the accessed memory line (*Directory entry* columns) and the state for the cache line that keeps a copy of the accessed memory line in each core (last 8 columns), **AFTER the execution of each** of the following memory accesses:

   (a) $core_0$ reads the contents of `v[0]`
   (b) $core_1$ reads the contents of `v[8]`
   (c) $core_0$ writes the contents of `v[0]`
   (d) $core_4$ reads the contents of `v[16]`

   Solution:

   (a) $core_0$ reads the contents of `v[0]`, which is not available in the associated cache (miss); $core_0$ generates a $PrRd$ event which activates the Snoopy protocol, placing a $BusRd$ transaction on the bus; since the local node and the home node are the same, and the initial status for the memory line in the directory is U (uncached), no coherence transactions are generated between NUMA nodes; the presence bits in the directory entry are set to 01 and state transitions to S (shared); a copy of the memory line is loaded into the local cache; and the cache line status in $core_0$ is updated from I (invalid) to S (shared) too.

Command	Coherence actions			Directory entry		State in cache associated to core							
	Core	Snoopy	Directory (yes/no)	Presence bits	State	0	1	2	3	4	5	6	7
$core_0$ reads v[0]	PrRd	BusRd	no	01	S	S	–	–	–	–	–	–	–
$core_1$ reads v[8]	PrRd	BusRd	no	01	S	S	S	–	–	–	–	–	–
$core_0$ writes v[0]	PrWr	BusUpgr	no	01	M	M	I	–	–	–	–	–	–
$core_4$ reads v[16]	PrRd	BusRd	yes	10	S	–	–	–	–	S	–	–	–

(b) $core_1$ reads the contents of `v[8]`, which is not available in the associated cache (miss); observe that `v[8]` resides in the same cache line as `v[0]`; $core_1$ generates a $PrRd$ event which activates the Snoopy protocol, placing a $BusRd$ transaction on the bus; again, the local node and the home node are the same, so, since no copies in other NUMA nodes exist, no coherence transactions between NUMA nodes are performed; presence bits and status in the directory entry do not change and a copy of the memory line is sent to the local cache of $core_1$, updating its status from I (invalid) to S (shared).

(c) $core_0$ writes the contents of `v[0]`, which is already loaded in the associated cache is S status (hit); $core_0$ generates a $PrWr$ event which activates the Snoopy protocol, placing a $BusUpgr$ transaction on the bus. Aa a result, the snoopy of $core_1$ invalidates its copy, setting its state to I; again, the local node and the home node are the same, so, since no copies in other NUMA nodes exist, no coherence transactions between NUMA nodes are performed; presence bits in the directory entry do not change but the status transitions to M (modified); similarly, the statues of the line in cache associated to $core_0$ also transitions to M.

(d) $core_4$ reads the contents of `v[16]`, which is not available in the associated cache (miss); observe that `v[16]` resides in a different memory line than `v[0]` and `v[8]`; $core_4$ generates a $PrRd$ event which activates the Snoopy protocol, placing a $BusRd$ transaction on the bus; now the local hub asks for a copy to the home node by sending a $RdReq$ message; the hub in the home NUMA node updates the information associated to the directory entry of the memory line, updating the presence bits to 10 to indicate that a copy exists in NUMA nodes 1 and status set to S (shared); then the hub sends a copy of the line to the local hub through a $Dreply$ message.; the state of the line in the cache associated to $core_4$ transitions from I (invalid) to S (shared).

4. Given the following code corresponding to a parallel region:

```
#pragma omp parallel num_threads(8)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int i_start = (N / howmany) * myid;
    int i_end = i_start + N / howmany;
    for (int i = i_start; i < i_end; i++)
        v[i] = comp (v[i]);  // comp does not perform any memory access
}
```

After executing the parallel region on the multiprocessor presented before, with all the assumptions in the previous question, the programmer observes that it does not scale as expected. We also know that: 1) $thread_i$ always executes on $core_i$, where $i = [0 - 7]$; 2) inside the function *comp* there are no memory accesses; and 3) vector `v` is the only variable that will be stored in memory (the rest of the variables will all be in registers of the cores). Which performance problem(s) does its execution have? Briefly justify your answer.

Solution: We can find mainly two performance problems:

(a) As the whole vector `v` is entirely allocated in MM_0, all the accesses to elements in the vector will be managed by NUMA node 0, generating a bottleneck and consequently degrading performance.

(b) The code above shows a well-balanced distribution of the vector `v` among the NUMA nodes of the systems. This implies that each thread is in charge of updating 64 elements / 8 threads = 8 consecutive elements. However, as a the line cache size is 64 bytes and that `sizeof (int)` = 4, then each cache line holds 64 bytes / 4 bytes = 16 elements, which means that a cache line will be shared by two threads, generating the so called False sharing performance problem.

PAR – Final Exam: Part 2 – Course 2020/21-Q1
January 18^{th}, 2021

Problem 1 Given two alternative versions to parallelise the execution of a code fragment:

version 1:

```
// x and l declared here
#pragma omp parallel num_threads(4)
#pragma omp single
{
#pragma omp taskloop grainsize(1)     // Ai
for (int i=0; i<4; i++) {
    l[i] = foo1(i);
    }

for (int i=0; i<4; i++) {
    #pragma omp task depend(inout: x) // Bi
    x += foo2(i, l[i]);
    }
#pragma omp taskwait

#pragma omp task                      // C
x += foo3(l);
}
```

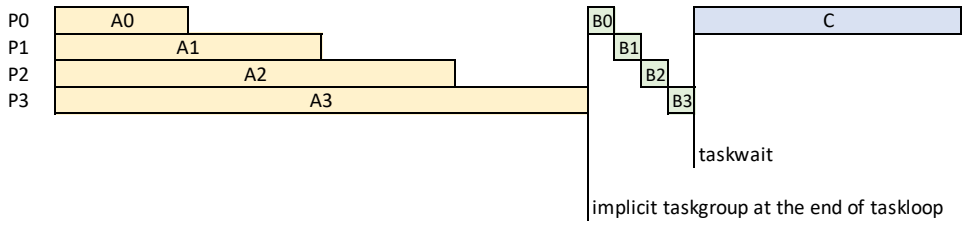
version 2:

```
// x and l declared here
#pragma omp parallel num_threads(4)
#pragma omp single
{
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])      // Ai
    l[i] = foo1(i);
    }

#pragma omp taskgroup task_reduction(+: x)
    {
    for (int i=0; i<4; i++) {
        #pragma omp task depend(in: l[i])   // Bi
                         in_reduction(+: x)
        x += foo2(i, l[i]);
        }

    #pragma omp task in_reduction(+:x)      // C
    x += foo3(l);
    }
}
```

Assume that 1) tasks generated in the first loop (tasks `Ai`) take $((i+1)\times 5)$ time units to execute; 2) tasks generated in the second loop (tasks `Bi`) take a constant time of 1 time unit to execute; 3) task `C` takes 10 time units to execute; 4) task creation and synchronisation overheads are negligible; and 5) the execution of functions `foo1`, `foo2` and `foo3` do not modify any global variable, i.e. their execution does not produce any data dependence.

1. (2 points) Draw a temporal diagram showing a possible execution on 4 processors for each code version above, obtaining the expression for its execution time T_4.

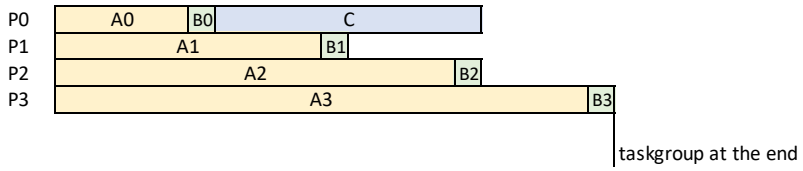
   In *version 1* dependences between tasks Ai and Bi are satisfied with the implicit `taskgroup` at the end of the `taskloop` region; dependences among tasks bi due to the sharing of global variable `x` are satisfied with task dependences using the `depend` clause; finally, the dependence between tasks Bi and C is satisfied with an explicit `taskwait`. A possible diagram for the parallel execution on 4 processors for this code, assuming that tasks are executed as soon as dependences are satisfied, could be this:



   which results in an execution time of $T_4 = 20 + 4 + 10 = 34$ time units.

   In *version 2* dependences between tasks Ai and Bi are satisfied with task dependences using the `depend` clause on each element of vector `l`; dependences among tasks Bi and `C` due to the sharing of global variable `x` are satisfied with task reductions in the scope of a `taskgroup` region, using `in_reduction` clause to specify the tasks that contribute to the reduction operation specified in the `task_reduction` clause of the `taskgroup`. A possible diagram for the parallel execution on 4

processors for this code, assuming that tasks are executed as soon as dependences are satisfied, could be this:



which results in an execution time of $T_4 = 20 + 1 = 21$ time units; observe that the execution of task C is totally hidden by the execution of the longest task B_3.

2. (1.5 points) As you know, the implementation of the `task_reduction` clause in the `taskgroup` construct that is used in *version 2* above requires that each task annotated with `in_reduction`, at the end of its execution, accumulates its local value for variable `x` (let's name it `xlocal`) into shared variable `x`. Write a code excerpt that implements the safe accumulation of the private variable `xlocal` into the global variable `x` using load linked and store conditional instructions:

   ```
   int load_linked (int *addr);
   int store_conditional (int *addr, int value);
   ```

   Recall that `store_conditional` returns 0 in case it fails or 1 otherwise. **Note:** You DON'T need to insert the sequence of instructions in *version 2* code above.

   The sequence of instructions that should be executed by each task to do the safe accumulation of its local variable `xlocal` into shared variable `x` could be:

   ```
   do {
       int tmp = load_linked(&x);
       tmp += xlocal;
   } while (store_conditional(&x, tmp)==0);
   ```

   Atomicity in the accumulation to variable `x` is guaranteed by the paired execution of `load_linked` and `store_conditional`, which is repeated in case it fails (i.e. when `store_conditional` returns 0), reading again the current value for `x`.

Problem 2 We have a code that explores the data stored in a hash table and creates a histogram:

```
#define CACHE_LINE_BYTES 64               // 64 bytes per cache line
#define HT_SIZE 1048576
#define NBINS 128

typedef struct {
  int value;
  elem * next;
} elem;

elem * HashTable[HT_SIZE], p;             // 8 bytes per pointer

int Histogram[NBINS], bin, i, value;      // 4 bytes per integer
...
for (i=0; i<HT_SIZE; i++) {
   p = HashTable[i];
   while ( p != NULL ) {
     value = p->value;
     bin = getbin(value, minval, maxval, NBINS);
     Histogram[bin]++;
     p = p->next;
   }
}
...
```

Let's assume that the minimum and maximum values stored in any element within the hash table are known and kept in variables `minval` and `maxval`, respectively. Routine `getbin` returns the *bin* where a `value` is classified according to the number of bins (`NBINS`) and `minval` and `maxval`. At the end of the computation each position in the histogram, a *bin* (or container), has to count the number of values found in the elements within the hash table for which function `getbin` returns the same value.

We ask you to write two parallel versions using OpenMP. **Note:** In both versions you just need to write the part of the code that shows clearly how the parallelization is done, writing "..." for the rest of the code.

1. (3 points) A *task decomposition* based on the use of **explicit tasks** which incurs low task management overhead. The solution must minimize synchronization overheads and maximize spatial locality in the accesses to vector `HashTable`.

   Solution: An alternative synchronization with locks has also been considered correct. We assume that the initial address of vector `HashTable` is aligned to a memory line.

```
#pragma omp parallel private(p,value,bin)
#pragma omp single
{
 int NT = omp_get_num_threads();
 int GS = HT_SIZE / NT;
 int CACHE_SIZE_ELEMS = CACHE_LINE_BYTES / sizeof(int *);
 int resize = (GS%CACHE_SIZE_ELEMS) ? CACHE_SIZE_ELEMS - (GS%CACHE_SIZE_ELEMS) : 0;
 GS = GS + resize;
 #pragma omp taskloop grainsize(GS)
 for (i=0; i<HT_SIZE; i++) {
    p = HashTable[i];
    while ( p != NULL ) {
      value = p->value;
      bin = getbin(value,minval,maxval,NBINS);
      #pragma omp atomic
      Histogram[bin]++;
      p = p->next;
    }
 }
}
```

2. (3.5 points) An *output block-cyclic geometric data decomposition* using **implicit tasks**. The solution should avoid *false sharing*.

   Solution: The solution requires an *if* used to check if the thread has to update a range of positions in the histogram. No synchronizations are required. We assume that the initial address of vector `Histogram` is aligned to a memory line. Granularity must be properly defined to avoid false sharing in the access to `Histogram`.

```
#pragma omp parallel private(bin,i,p,value)
{
 int thid = omp_get_thread_num();
 int NT = omp_get_num_threads();
 int CACHE_SIZE_ELEMS = CACHE_LINE_BYTES / sizeof(int);
 for (i=0; i<HT_SIZE; i++) {
    p = HashTable[i];
    while ( p != NULL ) {
      value = p->value;
      bin = getbin(value,minval,maxval,NBINS);
      if ( (bin/CACHE_SIZE_ELEMS)%NT == thid )
        Histogram[bin]++;
      p = p->next;
    }
 }
}
```

PAR – Final Exam – Course 2020/21-Q2
June 16^{th}, 2021

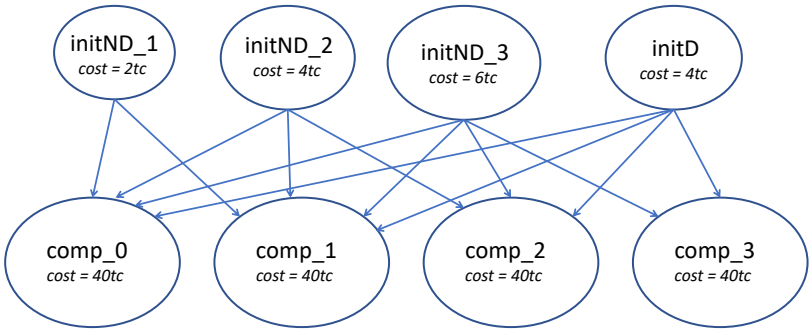
Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#define N 4
int A[N][N], B[N][N];
...
// initialization of non-diagonal elements
for (i=1; i<N; i++) {
    sprintf(stringMessage, "initND_%d", i);
    tareador_start_task (stringMessage);
    for (k=0; k<i; k++) {
        A[i][k] = init(i,k); // inner loop body cost = 2*tc
        A[k][i] = A[i][k];
    }
    tareador_end_task (stringMessage);
}

// initialization of diagonal elements
tareador_start_task ("initD");
for (i=0; i<N; i++) A[i][i] = init (i,i); // inner loop body cost = 1*tc
tareador_end_task ("initD");

// computation phase
for (i=0; i<N; i++) {
    sprintf(stringMessage, "comp_%d", i);
    tareador_start_task (stringMessage);
    for (k=0; k<N; k++) B[i][k] = foo (A[i][k]);  // inner loop body cost = 10*tc
    tareador_end_task (stringMessage);
}
```

We ask you to answer the following questions:

1. Draw the Task Dependence Graph (TDG) assuming the given value for constant `N` and the *Tareador* task definitions in the program. In the TDG, annotate each node with the name of the corresponding tasks (`initND_i`, `initD`, `comp_i`) and its cost.

   Solution:



2. Compute the T_1, T_∞ and P_{min} metrics associated to the TDG obtained in the previous question.

   Solution:

   The sum up of the cost of all the tasks determines $T_1 = 176 \times tc$. The critical path is composed of nodes: `initND_3` and `comp_X`, where X is any number between 0 and 3. Its cost is $T_\infty = 46 \times tc$. The minimum number of processors to achieve T_∞ execution time is $P_{min} = 4$.

3. Determine the assignment of tasks to processors that would yield the best *speed-up* on 4 processors. Calculate T_4 and S_4.

   Solution:

   Since the number of requested preocessors is $P = P_{min} = 4$, then $T_4 = T_\infty = 46 \times tc$, and therefore $S_4 = 176 \ / \ 46 = 3.8$ coincides with the *Parallelism* metric. The assignment of tasks to 4 processors is straightforward:

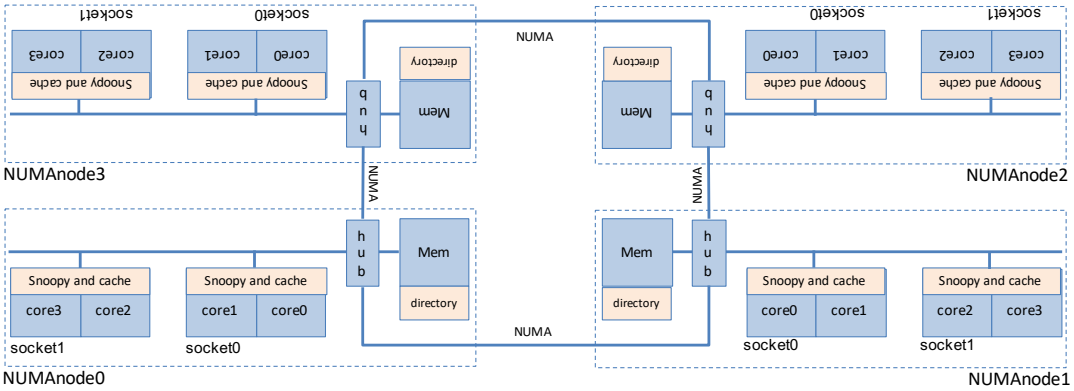
   P_0 = `initND_1, comp_0`

   P_1 = `initND_2, comp_1`

   P_2 = `initND_3, comp_2`

   P_3 = `initD, comp_3`

Problem 2 (1 point) Given the following NUMA system with 4 NUMA nodes, each NUMA node with 2 sockets sharing the access to node memory, and each socket with two cores sharing the access to a per-socket cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing the simplest write–invalidate MSI explained in class. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing the simplest write–invalidate MSU explained in class.



Assume that the home node for the line containing variable `var` is NUMAnode0, and at a given time there exist 3 clean copies of that line in cache memories: in socket0 in NUMAnode0, in socket0 in NUMAnode1 and in socket0 in NUMAnode2. Considering that the following memory accesses are done one after the other: 1) core2 in NUMAnode0 reads `var`; 2) core0 in NUMAnode3 reads `var`; and 3) core0 in NUMAnode3 writes `var`. **We ask you to select ONLY the eight sentences that you consider correct from the list below** (labeled from a) to o)). Each correct selection adds 0.125 points; each wrong selection subtracts 0.0625 points; if you select more than 8, only the first 8 will be considered; the grade for this problem is always in the range 0–1.

1. When core2 in NUMAnode0 reads variable `var`, which of the following sentences are correct?

   (a) Core2 issues PrRd.

   (b) The snoopy in socket1 issues BusRd on its local bus.

   (c) The snoopy in socket0 observes the BusRd command and places the line on the bus (Flush).

   (d) The hub associated to NUMAnode0 updates the directory for the line containing `var` to indicate that a new copy of the line exists inside NUMAnode0.

   (e) No coherence requests are sent to the rest of the NUMA nodes in the system.

   Solution: True, True, False, False, True

2. Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?

   (f) The snoopy in socket0 issues BusRd on its local bus.

   (g) The hub associated to NUMAnode3 finds the closest NUMA node that has a copy of the line and sends a RdReq to that NUMA node.

   (h) The hub of the NUMA node receiving the RdReq reads the line from the cache memory that is storing it.

   (i) NUMAnode3 receives a Dreply command with the line containing variable `var` and stores a copy in its main memory.

   (j) At the end the directory in the home NUMA node is updated so that all bits in the sharers list are set to 1 and the state is kept as S

   Solution: True, False, False, False, True

3. Finally, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?

   (k) The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.

   (l) The hub in NUMAnode3 issues an Invalidate command, going to the home NUMA node.

   (m) The home NUMA node checks if there are copies of the line in other NUMA nodes, sending to each one of them an Invalidate command.

   (n) The state for the line in the caches of the remote nodes receiving the Invalidate command (as well as in the home node) changes from S to I to indicate that the line is nor valid anymore.

   (o) At the end the directory in the home NUMA node only has bit 3 in the sharers list set to 1 and the state set to M.

   Solution: False, False, True, True, True

Problem 3 (3 points) Assume the following sequential code and $N >= 2$ and power of two:

```
#define N (1<<29) // A power of 2 value
typedef struct {
  float max; float min;
} min_max_t;

min_max_t find_min_max_it(float *v, int n) {
     min_max_t min_max;
     float max_duration = v[1]-v[0];
     float min_duration = v[1]-v[0];

     for (int i=2; i<n; i++) {
        float d = v[i] - v[i-1];
        if (d > max_duration) max_duration = d;
        if (d < min_duration) min_duration = d;
     }

     min_max.max = max_duration;
     min_max.min = min_duration;
     return min_max;
}

min_max_t find_min_max_rec(float *v, int n) {
    min_max_t min_max, min_max1, min_max2;

    if (n==2) {
        min_max.max = v[1]-v[0]; min_max.min = v[1]-v[0];
```

```
    } else {
        int n2 = n/2; // n is power of 2
        min_max1 = find_min_max_rec(v, n2);
        min_max2 = find_min_max_rec(v+n2, n2);

        if (min_max1.min < min_max2.min) min_max.min = min_max1.min;
        else min_max.min = min_max2.min;
        if (min_max1.max > min_max2.max) min_max.max = min_max1.max;
        else min_max.max = min_max2.max;
    }
    return min_max;
}

int main() {
    min_max_t min_max_V1, min_max_V2;
    int v1[N], v2[N];
    min_max_V1 = find_min_max_it(v1, N);
    min_max_V2 = find_min_max_rec(v2, N);
}
```

We ask you to answer the following independent questions:

1. Implement an OpenMP parallel version of function `find_min_max_it` and modify the main program as you consider to create an efficient iterative linear task decomposition version of the code. This implementation should avoid synchronizations within the loop and exploit the parallelism with a grainsize bigger than one iteration per task. You are ONLY allowed to use explicit tasks.

   Solution:

   The key points are :

   - Usage of taskloop construct with a granularity bigger than 1, for instance, 2 (`grainsize(2)`) or evenly distributing the iterations among threads (`num_tasks(omp_get_num_threads())`).
   - Usage of reduction clause to avoid synchronizations to update the max and min durations.
   - Add constructs parallel and single in the main program.

```
min_max_t find_min_max_it(float *v, int n) {
     min_max_t min_max;
     float max_duration = v[1]-v[0];
     float min_duration = v[1]-v[0];

     /* Assuming the number of threads is smaller than n */
     #pragma omp taskloop num_tasks(omp_get_num_threads()) \
                          reduction(max:max_duration)      \
                          reduction(min:min_duration)
     for (int i=2; i<n; i++) {
        float d = v[i] - v[i-1];
        if (d > max_duration) max_duration = d;
        if (d < min_duration) min_duration = d;
     }

     min_max.max = max_duration;
     min_max.min = min_duration;
     return min_max;
}

int main() {
    min_max_t min_max_V1, min_max_V2;
    float v1[N], v2[N];
```

```
    #pragma omp parallel
    #pragma omp single
    min_max_V1 = find_min_max_it(v1, N);

    min_max_V2 = find_min_max_rec(v2, N);
    }
}
```

2. Implement an OpenMP parallel version of function `find_min_max_rec` and modify the main program as you consider to create an efficient recursive task decomposition version of the code. This implementation should reduce the parallelization overheads due to the generation of tasks controlling it by the depth of the recursivity tree (MAX_DEPTH).

   Solution:

   The key points are :

   - Implement a recursive tree task decomposition
   - Add a new parameter to count the depth level
   - Usage of final clause and omp_in_final intrinsic to implement the cut-off based on the recursivity tree level. Note that it can be implemented using if statements controlling if the maximum depth is reached or not.
   - Force `min_max1` and `min_max2` to be shared, and a taskwait to wait for the two created tasks to be finished.
   - Add constructs parallel and single in the main program.

```
min_max_t find_min_max_rec(float *v, int n, int depth) {
    min_max_t min_max, min_max1, min_max2;

    if (n==2) {
        min_max.max = v[1]-v[0]; min_max.min = v[1]-v[0];
    } else {
        int n2 = n/2; // n is power of 2
        if (!omp_in_final()) {
            #pragma omp task shared(min_max1) final(depth>=MAX_DEPTH)
            min_max1 = find_min_max_rec(v, n2, depth+1);
            #pragma omp task shared(min_max2) final(depth>=MAX_DEPTH)
            min_max2 = find_min_max_rec(v+n2, n2, depth+1);
            #pragma omp taskwait
        } else {
            min_max1 = find_min_max_rec(v, n2, depth+1);
            min_max2 = find_min_max_rec(v+n2, n2, depth+1);
        }

        if (min_max1.min < min_max2.min) min_max.min = min_max1.min;
        else min_max.min = min_max2.min;
        if (min_max1.max > min_max2.max) min_max.max = min_max1.max;
        else min_max.max = min_max2.max;
    }
    return min_max;
}

int main() {
    min_max_t min_max_V1, min_max_V2;
    int v1[N], v2[N];
    min_max_V1 = find_min_max_it(v1, N);
```

```
        #pragma omp parallel
        #pragma omp single
        min_max_V2 = find_min_max_rec(v2, N, 0);
    }
```

Problem 4 (4 points) Assume the following sequential code fragment implementing a certain computation with matrix `out_matrix` and vector `in_vector`:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
double in_vector[N];
double out_matrix[M][M];
...
int i, row;
...
for (i = 0; i < N; i++) {
    row = random(M); // random returns a random number between 0 and M-1
    update_row(row, in_vector[i]);
}
```

The following code implements a parallel version for the above loop that uses the so called *"master–worker" paradigm*. In the *"master–worker" paradigm* the "master" thread (only one, thread P in the code below) is the only responsible for assigning work to the "worker" threads (P threads, numbered from 0 to P-1 in the code below, assuming a parallel region executed with P+1 processors). Communication between the "master" thread and a "worker" thread `k` is done through one element of vector `port`, in particular `port[k]`. Through this port `port[k]` the master sends to worker `k` the rows that it has to compute, one after the other, following a specific **output data decomposition** strategy:

```
#define N ... // number of elements in the input vector
#define M ... // number of rows and columns in the output matrix
#define P ... // number of worker threads
double in_vector[N];
double out_matrix[M][M];

typedef struct {
    int row;
    double value;
} Port;
Port port[P];

int i, row, destination;
...
#pragma omp parallel num_threads(P+1)
if (omp_get_thread_num() == P) {
    for (i = 0; i < N; i++) {
        row = random(M); // random returns a random number between 0 and M-1
        destination = thread_to_be_assigned(row, M, P);     // Question 4.1
        port[destination].row= row;
        port[destination].value= in_vector[i];
    }
} else {
    myid = omp_get_thread_num();
    for ( ; ; ) {
        update_row(port[myid].row, port[myid].value);
    }
}
```

The previous code is not complete since the master and worker threads need some sort of synchronization to ensure the proper assignment of work from master to worker threads. However, you SHOULD NOT WORRY about this issue by now and will address it later.

We ask you to:

1. Implement 3 versions of function `int thread_to_be_assigned(int row, int num_rows, int num_procs)` to implement a:

   (a) *BLOCK* data decompositon, assuming that M is a multiple of P;
   Solution:

   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       int num_elems = num_rows / num_procs;
       return (row / num_elems);
   }
   ```

   (b) *CYCLIC* data decomposition;
   Solution:

   ```
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return (row % num_procs);
   }
   ```

   (c) *BLOCK-CYCLIC* data decompositon, with block size `BS`;
   Solution:

   ```
   #define BS ...
   int thread_to_be_assigned(int row, int num_rows, int num_procs) {
       return ((row / BS) % num_elems);
   }
   ```

To address the synchronization issue between master and worker threads mentioned before the programmer is proposing to add a new field `ready` to the definition of `Port`, initially set to 0, and two new functions `wait4worker` and `wait4master`, as follows:

```
typedef struct {
    int ready;
    int row;
    double value;
} Port;

Port port[P];

void wait4worker (int num) {
    while (port[num].ready == 1);
    port[num].ready = 1;
}

void wait4master (int num) {
    while (port[num].ready == 0);
    port[num].ready = 0;
}
```

2. Modify the implementation of function `wait4worker` so that its execution is performed atomically (i.e. the read/write of `port[num].ready` is performed atomically), making use of the following atomic primitive:

   - `int test_and_set(int *addr)`: returns the value stored at the memory address pointed by `addr` and sets it to 1;

Solution:

```
void wait4worker (int num) {
    while (test_and_set(&port[num].ready) == 1);
}
```

Of course, solutions implementing a *test-test&set* solution have also been considered valid.

3. Similarly, modify the implementation of function `wait4master` so that its execution is performed atomically (i.e. ensuring atomicity in the read/write of `port[num].ready`), making use of the following atomic primitives:

   - `int load_linked (int *addr)`: returns the value stored at the memory address pointed by `addr`;
   - `int store_conditional (int *addr, int value)`: tries to write `value` into the memory address pointed by `addr`, returning 1 if it succeeds (no intervening store to that address has taken place since the last call to `load_linked` with the same memory address) or 0 if it fails.

   Solution:

```
void wait4master (int num) {
    do {
        while (load_linked(&port[num].ready) == 0);
    } while (store_conditional(&port[num].ready, 0) == 0);
}
```

You can assume that functions `test_and_set`, `load_linked` and `store_conditional` are compatible in terms of atomicity. **You do not have to modify the original parallel code to make it correct using functions `wait4worker` and `wait4master`, you simply need to implement an atomic version of these two functions.**

Finally, the programmer wants to avoid the possibility of having false sharing when accessing vector `port`.

4. Why false sharing may happen when accessing to vector `port`? Redefine the last definition of data structure `Port` to ensure that false sharing will not occur, assuming that `int` and `double` data types occupy 4 and 8 bytes, respectively, and that cache and memory lines are 64 bytes long,

   Solution: False sharing may occur since several consecutive elements of vector `port` can reside in the same cache line and each of them read/written by a different processor. The solution would be to make sure each element occupies a complete cache line. Since the structure `Port` includes 2 integer and 1 double, in total 16 bytes, we can add padding for a total of $64 - 16$ bytes, that is 48 bytes (which are occupied for example by 12 integer elements):

```
typedef struct {
    int ready;          // 4 bytes
    int row;            // 4 bytes
    double value;       // 8 bytes
    int padding[12];    // 48 bytes
} Port;                 // 64 bytes
Port port[P];           // 64 bytes per element
```

   Another option would be to convert `port` to a matrix, so that each row occupies a complete cache line. To achieve that, since the structure occupies 16 bytes, we need 4 elements in each row:

```
typedef struct {
    int ready;          // 4 bytes
    int row;            // 4 bytes
    double value;       // 8 bytes
} Port;                 // 16 bytes
Port port[P][64/16];    // 64 bytes per row
```

   and then modify all accesses in the code accordingly to only access to column 0, for example: `port[destination][0]` `row`.

PAR – Final Exam – Course 2021/22-Q1

January 17^{th}, 2022

Problem 1 (2.5 points)

The following code computes matrix `u[N][N]` by blocks of $BS \times N$ elements, with `N` very large:

```
double u[N][N];

// Compute elements in a block of BS x N elements
void compute_row_block(int ii) {
    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=1; j<N-1; j++) {
            double tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            u[i][j] = tmp/4;
        }
}

void main() {
    int NB = 2*P;          // Total number of row blocks
    int BS = N/NB;         // Number of rows per block

    // EVEN loop: traversing all EVEN blocks
    for (int ii=0; ii<NB; ii+=2)
        compute_row_block(ii*BS);

    // ODD loop: traversing all ODD blocks
    for (int ii=1; ii<NB; ii+=2)
        compute_row_block(ii*BS);
}
```

In this code the computation is divided in two parts: the so called *EVEN* loop computing half of the blocks first (blocks 0, 2, ...), and the so called *ODD* loop computing the other half of the blocks later (blocks 1, 3, ...). Before answering the first question below, think about the parallel execution opportunities in this code when defining each iteration of the *EVEN* and *ODD* loops as a task. Could all the tasks in the *EVEN* loop be executed in parallel? Could all the tasks in the *ODD* loop be executed in parallel? And could the execution of tasks in the *ODD* loop be performed in parallel with the execution of tasks in the *EVEN* loop?. Having all that in mind, **we first ask you to:**

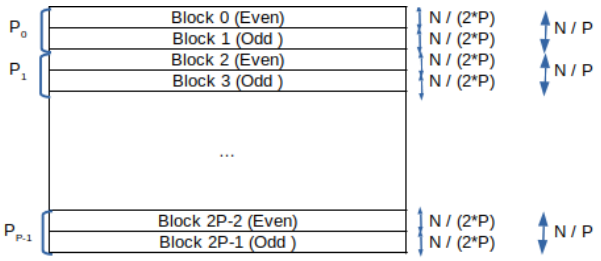
1. Write the expression for the contribution to the total parallel time T_P coming from the parallel computation time T_P^{comp} on an ideal machine with P processors, assuming that: 1) $NB = 2 * P$ perfectly divides the problem size `N`; 2) the execution time for a single iteration of the innermost loop body in routine `compute_row_block` takes t_c time units; and 3) no parallelisation overheads should be considered at this point.

   Solution:

   Since N is very large, we can consider $N - 2 \approx N$. Then,

   $T_P^{comp} = 2 \times BS \times N \times t_c = N^2/P \times t_c$

   Explanation:



All the tasks in the *EVEN* loop can be executed in parallel. All the tasks in the ODD loop can also be executed in parallel. However, tasks in the *ODD* loop can only be executed once the tasks in the EVEN loop computing the surrounding blocks have been completed. Each of the two loops iterate P times. Therefore, with P processors each processor will execute only 1 iteration of each loop. Since N is large, we can consider $N - 2 \approx N$. Thus, each processor computes N/P rows of size N, for a total of N^2/P executions of the innermost loop of routine `compute_row_block`. Note that half of them correspond to its *EVEN* block and the other half to its *ODD* block.

Next we consider that the ideal machine has the memory distributed among the P processors, In that machine, matrix `u` is divided row-wise in $NB = 2 * P$ blocks, where each block contains $BS = N/(2 * P)$ consecutive rows. Pairs of consecutive blocks are assigned to the same processor, so for example blocks 0 and 1 are owned by processor 0, blocks 2 and 3 are owned by processor 1; and so on and so forth. Each processor is in charge of computing all the rows in the blocks that are assigned to it, and therefore it will have to execute one iteration of the *EVEN* loop and one iteration of the *ODD* loop. Before answering the second question below, and having in mind the assignment of blocks and iterations, think about the need for processors to perform any remote access before starting the execution of tasks in the *EVEN* loop, or before starting the execution of tasks in the *ODD* loop; and, if affirmative, how many elements need to be transferred in each case and which processors to do them. Having all that in mind, next **we ask you to:**

2. Write the expression for the contribution to T_P coming from the data sharing overheads $T_P^{data\_sharing}$, if any, as part of the overall $T_P = T_P^{comp} + T_P^{data\_sharing}$. For that you should consider the data sharing model explained in class. Accesses to local memory are performed with zero overhead; accesses to remote memory take $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start–up time for the remote access and transfer time of one element, respectively. At a given moment, a processor can only perform one remote memory access to another processor, and can only serve a remote memory access from another.

   Solution:

   $T_P^{data\_sharing} = 2 \times (t_s + N \times t_w)$

   Explanation:

   In general, before an EVEN block can be computed a processor needs to perform a remote access to the N elements in the last row of the block above, which is owned by the previous processor (except for processor 0). No remote access is required to access the first row of the next block, because it is an ODD block owned by the same processor. All the processors can perform such remote accesses in parallel.

   An ODD block can only be computed once its surrounding EVEN blocks are computed. Before an ODD block can be computed a processor needs to perform a remote access to the N elements in the first row of the block below, which is owned by the next processor. No remote access is required to access the last row of the previous block, because it is an EVEN block owned by the same processor. All the processors can perform such remote accesses in parallel.

Problem 2 (2.5 points)

Given the following code fragment:

```
#define CACHE_LINE_SIZE  128
#define DBSize           (32*1024*1024)
#define NTHREADS         3

int count[NTHREADS];
int DB[DBSize];
int key;

// Initialization loop
for (int i = 0; i < NTHREADS; i++)
    count[i] = 0;
```

```
#pragma omp parallel num_threads(NTHREADS)
{
    int my_id = omp_get_thread_num();
    for (int i = my_id; i < DBSize; i += NTHREADS) {
        // read access to DB[i]
        if (DB[i]==key)
            // read access to count[my_id] followed by a write access
            count[my_id] = count[my_id] + 1;
    }
}
```

Assume a shared–memory UMA parallel system with 3 processors, each one with its own (private) cache memory. Data coherence in the system is maintained using Write Invalidate `MSI` protocol, with a Snoopy attached to each cache memory. Also assume 1) empty caches at the beginning of the program; 2) `count[0]` and `DB[0]` are aligned to the beginning of different memory lines; 3) the rest of variables are stored in registers; and 4) the size for a variable of type `int` is 4 bytes and cache line size is 128 bytes. **We ask you:**

1. Assume thread 0, running on processor 0, starts executing the sequential part of the program until the parallel region starts (i.e. it executes the initialization loop). How many cache lines are used to store the full `count` vector after the execution of the initialization loop? What is the cache coherence state for each cache line storing elements of `count` and in which private cache it is located?

   Solution:

   We only require one cache line to store the three elements of the vector `count`. The coherence state of the cache line will be M (Modified) and the memory cache that will store the vector is MC_0.

2. Assume the cache states in previous question (after the execution of the initialization loop) and that each thread i runs on processor P_i of the UMA system within the parallel region. Complete the table in the provided answer sheet which represents the first sequence of actions (from top to bottom of the table) executed by the three threads in the parallel region for the first iterations of the loop. State MC_i in each row has to be filled with the state of the cache line after the memory access in the action. For the rest of columns, you should specify the processor event ($PrRd_i$, $PrWr_i$), if there is cache hit or miss, the bus command ($BusRd_i$, $BusRdX_i$, $BusUpgr_i$), and the flush transaction ($Flush_i$), where i is the number of the processor where it is generated. Note: Observe that if you want to leave a cell empty, you can write "-".

   Solution:

Parallel Region Execution							
Action	CPU event	Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 read DB[0]	PrRd_0	miss	BusRd_0	-	S	-	-
P_0 read count[0]	PrRd_0	hit	-	-	M	-	-
P_1 read DB[1]	PrRd_1	miss	BusRd_1	-	S	S	-
P_1 read count[1]	PrRd_1	miss	BusRd_1	Flush_0	M->S	S	-
P_0 write count[0]	PrWr_0	hit	BusUpgr_0	-	S->M	S->I	-
P_1 write count[1]	PrWr_1	miss	BusRdX_1	Flush_0	M->I	I->M	-
P_2 read DB[2]	PrRd_2	miss	BusRd_2	-	S	S	S
P_2 read count[2]	PrRd_2	miss	BusRd_2	Flush_1	I	M->S	S
P_2 write count[2]	PrWr_2	hit	BusUpgr_2	-	I	S->I	S->M
P_0 read DB[3]	PrRd_0	hit	-	-	S	S	S
P_1 read DB[4]	PrRd_1	hit	-	-	S	S	S

3. Assuming the amount of data we are accessing in the program, a system with a main memory of 32 GBytes and 32 Kbytes of private cache memory per processor, what is the total number of bits that an UMA system would use to keep the cache coherence per processor and the overall system? Would this number of bits change if we change the protocol from `MSI` to `MESI`?

   Solution:

   Per processor/cache:
   $32kbytes \times \frac{1line}{128bytes} \times \frac{2bits}{1line} \rightarrow \frac{2^{15} \times 2}{2^7} bits \rightarrow 2^9 bits \rightarrow 512bits$

   Overall:
   3×512 bits devoted to coherence

   The number of bits would not change because we can still use 2 bits to represent 4 states (M, E, S, I) as when we have 3 states (M, S, I).

Problem 3 (2.5 points)

Given the following sequential program that computes the number of times the value stored in variable `element` appears in a vector of lists `data`:

```
#define NUM_ELEMS 10000
typedef struct list {
    int elem;
    struct list * next;
} list; // the basic component of a list

list * data[NUM_ELEMS]; // vector of lists, each list with varying number of elements
int element, count = 0; // value to search within data and number of times it appears

// function that returns the number of times element appears in theList
int list_search(list * theList, int element);

void main() {
    ...
    for (int entry = 0; entry < NUM_ELEMS; entry++) {
         int tmp = list_search(data[entry], element);
         count += tmp;
    }
    ...
}
```

Each of the lists may have a different number of elements, so when parallelising the program one should take care of load balancing. A parallel version for the sequential program above is also available, in which the original `for` loop has been substituted by a parallel region that assigns individual iterations to explicit tasks in such a way that tasks are generated under certain circumstances. One new shared variable `active_tasks` and two functions to operate it have also been added:

```
...
int active_tasks = 0;
// Functions to atomically add or subtract 1 to memory location pointed by address.
// The operation is saturated to the max or min value (i.e. the result can not be
// greater than max and smaller than min, respectively). They return value in memory
// location before operation
int atomic_inc(int *address, int max);
int atomic_dec(int *address, int min);

void main() {
    #pragma omp parallel
    #pragma omp single
    {
```

```
        int workers = omp_get_num_threads() - 1; // one thread focuses on
                                                 // task creation, the rest execute tasks
        for (int entry = 0; entry < NUM_ELEMS; entry++) {
            while (atomic_inc(&active_tasks, workers) == workers);
            #pragma omp task depend(inout: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
    }
}
```

After compiling the parallel program and executing it with P processors (with $P > 1$) we detect that the program **is not achieving any speed–up**, although it produces a correct result.

1. Assuming the functionality for functions `atomic_inc` and `atomic_dec` explained in the code itself, what are these two functions used for in the program? Which is the number of implicit and explicit tasks that are generated during the execution of the program?

   Solution:

   The program generates P implicit tasks, one for each thread in the `parallel` construct. Only one of them (the one entering in `single`) is in charge of generating the explicit tasks: one explicit task is generated for each iteration of the for loop, so in total `NUM_ELEMS` explicit tasks. Functions `int atomic_inc(int *address, int max)` and `int atomic_dec(int *address, int min)` are used to control the number of tasks generated (kind of cut-off mechanism), in such a way that no more than $P - 1$ explicit tasks are simultaneously pending to execute or executing.

2. In the parallel version provided above, how many of these explicit tasks can be simultaneously executing? Rewrite the program above making the minimum appropriate changes in order to increase this number and, as a consequence, achieve a much better speed–up. Make sure the program generates the correct result. After these minimal changes, which can be the maximum number of explicit tasks that could be simultaneously executing?

   Solution:

   The `depend(inout:count)` clause used in `task` forces explicit tasks to execute sequentially, in the order they are created. This is not the most appropriate way to guarantee the race condition in this program when updating variable `count`; instead the use of `atomic` would enable the parallelism in the execution of multiple invocations to `list_search` while guaranteeing the correct update of variable `count`. With this change, a maximum of $P - 1$ tasks could be executing simultaneously.

   ```
   #pragma omp task // shared(count) and firstprivate(entry) by default
       {
       int tmp = list_search(data[entry], element);
       #pragma omp atomic
       count += tmp;
       atomic_dec(&active_tasks, 0);
       }
   ```

   Alternatively, a solution based on task reductions would also be valid:

   ```
   #pragma omp taskgroup task_reduction(+: count)
   for (int entry = 0; entry < NUM_ELEMS; entry++) {
       while (atomic_inc(&active_tasks, workers) == workers);
   ```

```
            #pragma omp task in_reduction(+: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
```

3. Do an implementation for function `atomic_inc` making use of load–linked (`ll`) and store–conditional (`sc`) operations, defined as follows:

```
int ll(int *address); // returns the value stored in address
int sc(int *address, int value); // stores value in address if atomicity with ll
                                 // has been accomplished, returning true (1);
                                 // retuns false (0) otherwise
```

Solution:

A possible solution could be:

```
int atomic_inc(int *address, int max) {
    int tmp = ll(address);
    while ((tmp < max) && !sc(address, tmp+1)) tmp = ll(address);
    return(tmp);
}
```

In this solution, if the value read from memory `address` is equal to `max`, then the while loop finishes, returning the value just read. If the value is smaller than `max`, then a `sc` of the incremented value on the same memory `address` is attempted; if it succeeds the while loop is finished, again returnin the original value read from memory. If `sc` fails, then the new value from memory `address` is read again.

An equivalent code written in a different (more explicit) way would be:

```
int atomic_inc(int *address, int max) {
    int retsc = 0;
    do {
        int tmp = ll(address);
        if (tmp == max) return(tmp);
        retsc = sc(address, tmp+1);
    } while (retsc == 0);
    return(tmp);
}
```

Another version that always writes to memory `address` would be:

```
int atomic_inc(int *address, int max) {
    int tmp, newvalue;
    do {
        tmp = newvalue = ll(address);
        if (tmp < max) newvalue++;
    } while (!sc(address, newvalue));
    return(tmp);
}
```

Observe that in this case if the value in memory `address` is already `max`, the same value `max` is unnecessarily written again to memory `address`.

4. Finally we found a recursive version alternative to the original sequential code:

```
...
void rec_list_search(list ** data, int size, int element) {
    int tmp = list_search(data[0], element);
    count += tmp;
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    rec_list_search(&data[0], NUM_ELEMS, element);
}
```

Write a parallel version for it that implements a *recursive* **leaf** *task decomposition*. This new version should not implement any cut-off mechanism to restrict the number of tasks that are generated.

Solution:

```
void rec_list_search(list ** data, int size, int element) {
    #pragma omp task shared(count)
    {
        int tmp = list_search(data[0], element);
        #pragma omp atomic
        count += tmp;
    }
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    rec_list_search(&data[0], NUM_ELEMS, element);
    ...
}
```

Problem 4 (2.5 points)

Given the following code fragment computing matrix `m[N][N]`, with `N` much larger than the number of processors P to be used in the parallel execution, and with `N` not necessarily a multiple of P:

```
telem m[N][N];

for (int i=1; i<N-1; i++)
   for (int j=0; j<N; j++)
       m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
```

We ask you to:

1. Decide the most appropriate *geometric data decomposition strategy* for matrix `m` and write a parallel version of the code above using OpenMP that corresponds to it. Your solution should a) minimize the synchronization overhead among implicit tasks and b) guarantee that the load unbalance is limited to `N` elements (i.e. the number of elements in a row or column of the matrix).

   Solution:

   There are dependencies between iterations of the `for-i` loop: a RAW dependency given by $m[i][j]$ to $m[i-1][j]$ and a WAR dependency given by $m[i][j]$ to $m[i][j+1]$ for i in $[2..N-2]$. There are

no dependencies between iterations of the `for-j` loop, so we can fully parallelize it, with no need for synchronization.

Consequently, it's advisable to choose a *Geometric Block Data decomposition* by columns. We must take care of adjusting the block size (number of columns of N elements each) so that there is at most 1 column of difference in size between the different blocks. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size. A *Geometric Cyclic Data decomposition* by columns would also be correct, as the amount of work would be balanced automatically, without any extra calculation.

```
telem m[N][N];
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int BS = N / P;
  int start = myid * BS;
  int end = start + BS;
  int mod = N % P;
  if (mod > 0) {
     if (myid < mod)
        start += myid;
        end = start + BS + 1;
     }
     else {
         start += mod;
         end += start + BS;
     }
  }
  for (int i=1; i<N-1; i++) {
    for (int j=start; j<end; j++) {
        m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
    }
  }
}
```

2. Now consider that the program is going to be executed on a parallel machine in which memory lines are 128 bytes long. The allocation in memory for matrix `m` is aligned to the start of a memory line and `sizeof(telem)` is 8 bytes (`N` is not necessarily multiple of `sizeof(telem)`). Decide the most appropriate *geometric data decomposition strategy* in this case and re-write the previous OpenMP parallel code and, if necessary, the definition of matrix `m`. Your solution should a) maximize parallelism among implicit tasks; and b) maximize data locality and reduce coherence traffic.

   Solution:

   In order to preserve data locality we must take care of the memory line size when accessing elements of the matrix. In this sense, given that `sizeof(telem) = 8 bytes`, a memory line of 128 bytes can hold up to 16 elements. For this reason we will consider only block sizes with values multiple of 16. Consequently we apply a *Geometric Block-cyclic data decomposition* by columns, with block size = 16. Given that N » P we can assume a load unbalance of one block (N x 16 elements). In addition, in case N is not multiple of `sizeof(telem)`, we must add padding at the end of the row to avoid generating false sharing with the beginning of the next row. In order to apply the owner compute rule, the distribution of the matrix `m` is by columns with the resulting block size.

```
#define MEMORYLINESIZE 128
#define BS MEMORYLINESIZE/sizeof(telem)
#define X (N%BS==0? 0: (BS - (N % BS)))
```

```
telem m[N][N+X];

int BS = MEMORYLINESIZE / sizeof(telem);
...
#pragma omp parallel num_threads(P)
{
  int myid = omp_get_thread_num();
  int start = myid * BS;
  int end = N;

  for (int i=1; i<N-1; i++) {
    for (int jj=start; jj<end; jj+=BS*P)
        for (int j=jj; j<j+BS; j++)
            m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
}
```

Student name: .....

Answer sheet for **Question 2.2**.

Parallel Region Execution							
Action	CPU event	Cache Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 reads DB[0]							
P_0 reads count[0]							
P_1 reads DB[1]							
P_1 reads count[1]							
P_0 writes count[0]							
P_1 writes count[1]							
P_2 reads DB[2]							
P_2 reads count[2]							
P_2 writes count[2]							
P_0 reads DB[3]							
P_1 reads DB[4]							