PAR – Final Exam Theory/Problems– Course 2024/25-Q1

January 16^{th}, 2025

Grade Publication January 22^{nd}, 2025 - Exam Review 23^{rd} - 16:00h–17:00h - C6E106

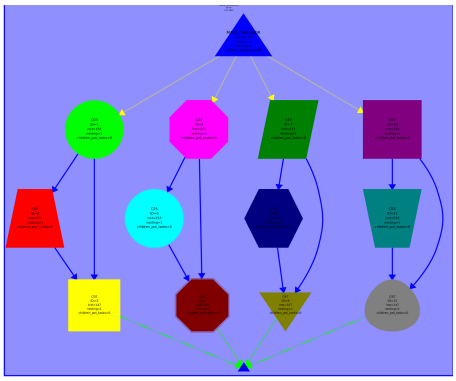
Problem 1 (2.5 points) Given the following C program instrumented with Tareador:

```
#define N 8
double M[N][N];

void main() {
char str[10];

  for (int ii=0; ii<N; ii+=2)
    for (int jj=3; jj<N; jj+=2) {
      sprintf(str,"C%d%d",ii,jj)
      tareador_start_task(str);
      for (int i=ii; i<ii+2; i++)
        for (int j=jj; j<min(jj+2,N); j++)
          M[i][j] = foo(M[i][j-1], M[i][j-2], M[i][j-3]);
      tareador_end_task(str);
    }
}
```

We ask you to:

1. Draw the TDG based on the above task definitions. Each task should be clearly labeled with the values for variables *ii* and *jj* at the time of task instantiation.

   Solution:



2. Consider that the cost of executing function `foo` is constant and equal to 10 t.u. and the cost of the rest of the code is negligible. Calculate T_1, P_{min} and T_∞.

   Solution:

   T1 = 400 t.u. (40*4+40*4+20*4) Pmin = 4 Tinf = T4 = 100 t.u. (40+40+20)

3. Consider now that the program is executed on a distributed memory architecture with 4 processors and that matrix M is initially stored in the memory assigned to P_0. Consider the data sharing model explained in class in which the latency of a remote memory access is $t_s + t_w \times m$, where t_s is 5 t.u. and t_w is 1 t.u. Function `foo` takes 10 t.u. as stated in the last question. We ask you to calculate T_4 assuming the best task scheduling policy. There is no need to move data back to P_0 once memory is modified.

   Solution:

   T_4 = 100 + 3x16 t.u. = 148 t.u.

   Tasks $C_{ii,jj}$ are assigned to processor $P_{ii/2}$. All tasks executing in P_0 have the data they need. Tasks $C_{2,3}$, $C_{4,3}$ and $C_{6,3}$ need data from P_0, the first 3 elements in each row to be processed. 3 elements has to be read from P_0 from two different rows, and therefore, two messages of 3 elements are required: $2 \times (t_s + 3 \times t_w)$, 16 t.u. Data has to be read sequentially from processor 0, and the critical path will be stablished by the time to receive the data at the last processor reading from processor 0 plus the time to process the three tasks assigned to that processor.

Problem 2 (2.5 points) A *convolution* is an important operation in machine learning and signal processing. A *filter* is a small matrix storing values useful for detecting some pattern in the input. Through the convolution the filter slides across the input data (a larger matrix) in a step-wise manner. At each step, an *element-wise multiplication and addition* is performed: at each position, the filter's values are multiplied by the corresponding values in the input data under it. These products are then summed to produce a single output value. The process is repeated across the entire input, generating an output matrix that highlights the presence of the filter's specific feature. Consider the following sequential program that performs a *convolution* using a 3×3 filter. The input matrix has a halo (initial and final row and column initialized to zeros). Thus, the input matrix has two more rows and columns than the output matrix:

```
#define PADDED_N(N) ((N) + 2) // Dimension of padded input matrix size including halo
void applyConvolutionIterative(int *input, int *output, int *filter, int N);

int main() {
    int N = 1000;               // Output matrix dimension
    int paddedN = PADDED_N(N);  //  Input matrix dimension

    // Allocate and initialize input and output matrices
    int *output = (int *) calloc(N * N, sizeof(int)); // initializes to 0's
    int *input  = (int *) malloc(paddedN * paddedN * sizeof(int));
    init_input_matrix(input);

    // Example of a 3x3 filter
    int filter[3 * 3] = { 0, 1, 0,     1, -4, 1,     0, 1, 0 };

    applyConvolutionIterative(input, output, filter, N);
    ...
    print_matrix(output);
    free(input); free(output);
    return 0;
}

void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;
```

```
    for (row = 1; row <= N; row++) {
        for (col = 1; col <= N; col++) {
            sum = 0;

            // Apply the 3x3 filter to a neighborhood of 3x3 elements in the input matrix:
            for (int fi = -1; fi <= 1; fi++) {
                for (int fj = -1; fj <= 1; fj++) {
                    int ni = row + fi; // Neighbor row index
                    int nj = col + fj; // Neighbor column index

                    int neighbor_idx = ni * PADDED_N(N) + nj; // Row-wise storage
                    int filter_idx = (fi + 1) * 3 + (fj + 1);
                    // Perform the element-wise multiplication and addition:
                    sum += input[neighbor_idx] * filter[filter_idx];
                }
            }
            output[(row - 1) * N + (col - 1)] = sum; // Row-wise storage
        }
    }
}
```

Note: When writing your solutions you do NOT need to write the whole code. Instead, write the minimum code excerpts that allows the reader to understand clearly how the parallelization has been done.

We ask you to: write a parallel version of routine `applyConvolutionIterative` using OpenMP's explicit tasks. The solution must try to exploit the maximum parallelism among threads while minimizing the data sharing synchronization and task creation overheads.

Solution: There are no loop-carried dependencies in this code. In order to be as efficient as possible: 1) work must be split by rows since row-wise storage is used for both the input and the output matrix: and 2) tasks must have an appropriate granularity, namely a chunk of approximately $\frac{N}{NT}$ consecutive rows being NT the number of threads being used, departing from a very fine grain granularity because no load unbalancing is expected since all iterations should take the same amount of associated work.

One possible solution using `#pragma omp taskloop`:

```
void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop private(col, sum) grainsize(N / omp_get_num_threads())
    for (row = 1; row <= N; row++) {
        for (col = 1; col <= N; col++) {
            sum = 0;

            // Apply the 3x3 filter
            ... // Not changed

            output[(row - 1) * N + (col - 1)] = sum;
        }
    }
}
```

Another possible solution using `#pragma omp task`:

```
void applyConvolutionIterative(int *input, int *output, int *filter, int N) {
    int row, col, sum;
```

```
    #pragma omp parallel
    #pragma omp single
      {
        int BS = N / omp_get_num_threads();
        for (int row_start = 1; row_start <= N; row_start += BS) {
            int row_end = (row_start + BS - 1 <= N) ? row_start + BS - 1 : N;
            #pragma omp task  private(row, col, sum)  // default firstprivate(row_start, row_end)
            for (row = row_start; row <= row_end; row++) {
                for (col = 1; col <= N; col++) {
                    sum = 0;

                    // Apply the 3x3 filter
                    ... // Not changed

                    output[(row - 1) * N + (col - 1)] = sum;
                }
            }
        }
      }
}
```

We have a recursive implementation which we also want to parallelize.

```
#define PADDED_N(N) ((N) + 2) // Dimension of padded input matrix size including halo
#define THRESHOLD 50          // Base case threshold

void applyConvolutionBaseCase(int *input, int *output, int *filter, int N,
                              int row_start, int row_end);

void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end);
int main() {
    // Same main as in the previous question except for the call to the recursive version

    ...
    // Apply convolution recursively
    applyConvolutionRecursive(input, output, filter, N, 1, N);
    ...
}

// Iterative base case for convolution
void applyConvolutionBaseCase(int *input, int *output, int *filter, int N,
                              int row_start, int row_end) {
    for (int row = row_start; row <= row_end; row++) {
        for (int col = 1; col <= N; col++) {
            int sum = 0;

            // Apply the 3x3 filter
            ... // Same as in the iterative code in the previous question.

            output[(row - 1) * N + (col - 1)] = sum; // Row-wise storage
        }
} }

// Recursive function to apply convolution
void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end) {
    if (row_end - row_start + 1 <= THRESHOLD) {
```

```
        applyConvolutionBaseCase(input, output, filter, N, row_start, row_end);
        return;
    }

    int mid = (row_start + row_end) / 2;

    applyConvolutionRecursive(input, output, filter, N, row_start, mid);
    applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end);
}
```

We ask you to: write a parallel version of routine `applyConvolutionRecursive` using OpenMP's explicit tasks. The solution must be efficient, exploiting parallelism as soon as possible but not on the leaves of the recursivity. In addition the code must be prepared to control the creation of tasks based on the depth of the recursivity and a value stored in a constant named MAX_DEPTH.

Solution: In order to exploit parallelism as soon as possible we have to implement a *Tree* decomposition. The solution implements *cutoff* based on the depth of the recursivity.

```
#define MAX_DEPTH 4           // Maximum depth for task creation

// Iterative base case for convolution is not changed at all
...

// Recursive function to apply convolution
void applyConvolutionRecursive(int *input, int *output, int *filter, int N,
                               int row_start, int row_end, int depth) {
    if (row_end - row_start + 1 <= THRESHOLD) {
        applyConvolutionBaseCase(input, output, filter, N, row_start, row_end);
        return;
    }

    int mid = (row_start + row_end) / 2;

    if ( !omp_in_final() ) {
        #pragma omp task final(depth >= MAX_DEPTH)
        applyConvolutionRecursive(input, output, filter, N, row_start, mid, depth + 1);

        #pragma omp task final(depth >= MAX_DEPTH)
        applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end, depth + 1);

    } else {
        applyConvolutionRecursive(input, output, filter, N, row_start, mid, depth + 1);
        applyConvolutionRecursive(input, output, filter, N, mid + 1, row_end, depth + 1);
    }
}


int main() {
    // Same main as in the previous question except for the call to the recursive version

    ...
    // Apply convolution recursively in parallel
    #pragma omp parallel
    #pragma omp single
    applyConvolutionRecursive(input, output, filter, N, 1, N, 0);
    ...
}
```

Problem 3 (2.5 points) Given the following code fragment computing matrix `A[N][N]`, an `N` much larger than the number of processors `P` (particularly `N » 4P`).

```
#define N .. // a value much larger than P
#define k N/4

float A[N][N], B[N][N];
...
for (int i = k; i < N-k; i++)
    for (int j = i; j < N-k; j++)
         A[i][j] = foo (A[i][j-k], A[i][j], B[i-k][j]); // computation function
```

We ask you to: Decide the most appropriate *Input/Output Geometric Data Decomposition* for matrices `A` and `B` and write a parallel version of the code by adding the *OpenMP* necessary directives and clauses, making use of only implicit tasks. Your solution proposal should:

- Minimize the synchronization overhead among implicit tasks
- Minimize the load unbalance
- Maximize data locality

Solution:

The best option is a geometric cyclic data decomposition of matrix A and matrix B by rows along the P processors, starting allocation on processor 0, of row k for matrix A, and of row 0 (i-k) for matrix B. This will help to exploit data locality and avoid data dependence synchronization.

```
#define N .. // a value much larger than P
#define k N/4

float A[N][N], B[N][N];
...
#pragma omp parallel num_threads(P)
{
 int myid = omp_get_thread_num();

 for (int i = k+myid; i < N-k; i += P)
    for (int j = i; j < N-k; j++)
         A[i][j] = foo (A[i][j-k], A[i][j], B[i-k][j]);
}
```

Surname, name .....

Problem 4 (2.5 points)

Assume the definition and allocation of matrices of the previous exercise. In addition, consider a shared-memory multiprocessor system composed by two NUMA nodes, each with two processors (cores) and 64 GBytes of shared main memory (MM). Each core has a private cache of 32 MBytes. Memory lines are 64 bytes long (one memory line per memory page), the allocation in memory for matrix `A` and matrix `B` are aligned to the start of a memory line and `sizeof(float)` is 4 bytes.

Cores 0 and 1 belong to NUMA node 0 and cores 2 and 3 are allocated in NUMA node 1. Data coherence is maintained using Write-Invalidate MSI protocols, with a Snoopy attached to each cache memory to provide coherency within each NUMA node and Write-Invalidate MSU directory-based coherence among the two NUMA nodes.

Assume now that matrix `A` is entirely allocated in the MM of Numa Node 0, and all cache memories of the multiprocessor system are empty, fill in the attached table with the sequence of processor commands (Core), bus transactions within NUMA nodes (Snoopy), transactions yes/no between NUMA nodes (Directory), the presence bits, state for each cache and memory line, to keep cache coherence, AFTER the execution of each of the following sequence of commands:

Memory access	Core	Coherence commands		Presence bits	State in MM	State in cache			
		Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_0$ reads `A[0][0]`									
$core_3$ reads `A[0][5]`									
$core_1$ writes `A[0][15]`									
$core_2$ writes `A[1][15]`									

Solution:

Memory access	Core	Coherence commands		Presence bits	State in MM	State in cache			
		Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_0$ reads A[0][0]	$PrRd_0$	$BusRd_0$	*no*	01	S	S	-	-	-
$core_3$ reads A[0][5]	$PrRd_3$	$BusRd_3$	*yes*	11	S	S	-	-	S
$core_1$ writes A[0][15]	$PrWr_1$	$BusRdX_1$	*yes*	01	M	I	M	-	I
$core_2$ writes A[1][15]	$PrWr_2$	$BusRdX_2$	*yes*	10	M	-	-	M	-

PAR – Final Exam Theory/Problems– Course 2023/24-Q2

June 11^{th}, 2024

Problem 1 (2.5 points) Given the following C program instrumented with `Tareador`:

```
#define BS  4
#define N   16
int ii, jj, i, j;
char stringMessage[128];
int M[N][N];

for (ii=0; ii<N; ii+=BS)
  for (jj=0; jj<N; jj+=BS) {
    sprintf(stringMessage,"CoP(%d,%d)",ii,jj);
    tareador_start_task(stringMessage);

    for (i=ii; i<min(ii+BS,N) ; i++)
      for (j=max(1,jj); j<min(jj+BS,N-1) ; j++)
        M[i][j]= M[i][j] + M[i][j-1] + M[i][j+1] + color_pixel(i,j); // 10 t.u.

    tareador_end_task(stringMessage);
   }
```

Assume each iteration of the innermost loop body takes 10 time units, `BS` and `N` are 4 and 16, respectively, only matrix `M` is stored in memory (rest of variables are in registers), and function `color_pixel` only computes a value function of the induction variables `i` and `j`. **We ask you to:**

1. Draw the TDG of the parallel strategy above. Indicate which is the cost of each task.

   Solution:

   ```
   CoP(0,0)  -> CoP(0,4)  -> CoP(0,8)  -> CoP(0,12)
    (120)        (160)        (160)        (120)

   CoP(4,0)  -> CoP(4,4)  -> CoP(4,8)  -> CoP(4,12)
    (120)        (160)        (160)        (120)

   CoP(8,0)  -> CoP(8,4)  -> CoP(8,8)  -> CoP(8,12)
    (120)        (160)        (160)        (120)

   CoP(12,0) -> CoP(12,4) -> CoP(12,8) -> CoP(12,12)
    (120)        (160)        (160)        (120)
   ```

2. Compute T_1, T_∞ and P_{min}.

   Solution:

   $T_1 = 4(120 + 160 + 160 + 120)t.u. = 2240t.u.$
   $T_\infty = 120 + 160 + 160 + 120t.u. = 560t.u.$
   $P_{min} = 4$

3. Write the expression that determines the execution time T_4, clearly indicating the contribution of the computation time T_4^{comp} and data sharing overhead T_4^{mov}, for the two following assignments of tasks to processors:

#proc	Assignment 1 (task id)	Assignment 2 (task id)
0	CoP(0,0), CoP(0,4), CoP(0,8), CoP(0,12)	CoP(0,0), CoP(4,0), CoP(8,0), CoP(12,0)
1	CoP(4,0), CoP(4,4), CoP(4,8), CoP(4,12)	CoP(0,4), CoP(4,4), CoP(8,4), CoP(12,4)
2	CoP(8,0), CoP(8,4), CoP(8,8), CoP(8,12)	CoP(0,8), CoP(4,8), CoP(8,8), CoP(12,8)
3	CoP(12,0),CoP(12,4),CoP(12,8),CoP(12,12)	CoP(0,12),CoP(4,12),CoP(8,12),CoP(12,12)

You can assume: 1) a distributed-memory architecture with 4 processors; 2) matrix `M`, is initially distributed by rows in Assignment 1 (N/BS consecutive rows per processor) and initially distributed by columns in Assignment 2 (N/BS consecutive columns per processor); 3) once the loop is finished, you don't need the return matrix to their original distribution; 4) data sharing model with communication time per message $t_{comm} = t_s + m \times t_w$, being t_s, m, t_w the start–up time, the number of elements, and transfer time of one element, respectively; 5) BS perfectly divides N; and 6) the execution time for a single iteration of the innermost loop body takes 10 t.u..

Assignment 1 Solution:

$T_4 = T_4^{comp} + T_4^{mov}$
$T_4^{mov} = 0$; All data is local. No communication is needed.
$T_4^{comp} = (2 \times ((BS-1) \times BS) + (\frac{N}{BS} - 2) \times (BS \times BS)) \times 10t.u.$;

Assignment 2 Solution:

$P = 4$;
$T_4 = T_4^{comp} + T_4^{mov}$

$T_4^{mov} = t_s + N \times t_w + (P-2) \times (t_s + BS \times t_w) + \frac{N}{BS} \times (t_s + BS \times t_w)$;

$T_4^{comp} = ((BS-1) \times BS + (P-2) \times (BS \times BS) + (\frac{N}{BS} - 1) \times (BS \times BS) + (BS-1) \times BS) \times 10t.u.$;

Note : Regarding to the T_4^{mov}, there is a initial communication of the right boundary (first column of next processor), that all processor but last one have to do. This communication can be done by all processors in parallel. Then, each processor (but first one) hast to read BS elements of the left boundary, produced by a task in previous processor (dependence).

Regarding to T_4^{comp}, the last row of tasks executed by processor $P-1$ has $BS \times BS - 1$ elements to process each, but this processor has to wait for $\frac{N}{BS} - 1$ tasks of previous processor that last each for $BS \times BS \times 10t.u.$, therefore, this time should be taken into account for the first three tasks of last processor. Then, it has to process $BS \times BS - 1$ elements of the very last task.

Problem 2 (1.5 points) Consider the following sequential program that performs a matrix multiplication ($C = A \times B$):

```
int A[L][M], B[M][N], C[L][N];
int main() {
   int l, n, m;
   ...
   // Matrix Mutiplication
   for (l=0; l<L; l++)
      for (n=0; n<N; n++)
         for (m=0; m<M; m++)
             C[l][n] += A[l][m]*B[m][n];
   ...
   // Output results
   for (l=0; l<L; l++)
      for (n=0; n<N; n++)
          printf("C[%d][%d]=%d\n",l,n,C[l][n]);
}
```

We ask you to:

1. Write an OpenMP version to parallelize `Matrix multiplication` computation code using an iterative task decomposition strategy where: 1) the granularity of the tasks should be 2 iterations of the middle loop (loop n); and 2) the synchronization overheads are minimized.

   A Solution:

   ```
   ...
   int A[L][M], B[M][N], C[L][N];

   int main() {
      int l, n, m;
      ...

      #pragma omp parallel private(l,m)
      #pragma omp single
        for (l=0; l<L; l++)
           #pragma omp taskloop grainsize(2) nogroup
           for (n=0; n<N; n++)
              for (m=0; m<M; m++)
                 C[l][n] += A[l][m]*B[m][n];
      ...

      for (l=0; l<L; l++)
         for (n=0; n<N; n++)
             printf("C[%d][%d]=%d\n",l,n,C[l][n]);

   }
   ```

   Note: Tasks are independent among them. Thus, it is not necessary to add any synchronization between tasks neither for solving data race conditions. On the other hand, it is important to add nogroup to avoid taskgroup sinchronization, that is not necessary and we want to minimize synchronization overheads.

2. Do you think parallelizing the `Output results` code can improve the performance of the program while obtaining the expected output? Justify briefly your answer.

A Solution: The output results code have to executed sequentially as the order of executing between the different `printf`'s have to be preserved. For this reason, there is no point on parallelizing it as it will not improve performance.

Problem 3 (1.5 points) Consider the following recursive program that copies an array into another one:

```
double X[N],Y[N];

int copy(double * __restrict__ input, double * __restrict__ output, int n) {
   if (n<=32)
      for (int i=0; i<n; i++) output[i] = input[i];
   else {
      copy(input, output, n/2);
      copy(input+n/2, output+n/2, n-n/2);
} }

int main() {
   ...
   copy(X,Y,N);
   ...
}
```

We ask you to: write a parallel OpenMP program that performs a parallel and efficient recursive task decomposition, reducing the task creation overheads and minimizing data sharing synchronizations. Note that `__restrict__` indicates that input and output are not overlapped in memory.

A Solution:

```
SOLUTION:
...
#define CUTOFF 5

double X[N],Y[N];

int copy(double *__restrict__ input, double * __restrict__ output, int n, int depth) {
   if (n<32)
       for (int i=0; i<n; i++) output[i] = input[i];
   else {
    if (!omp_in_final()) {
      #pragma omp task final(depth>=CUTOFF)
      copy(input, output, n/2, depth+1);
      #pragma omp task final(depth>=CUTOFF)
      copy(input+n/2, output+n/2, n-n/2, depth+1);
    } else {
      copy(input, output, n/2, depth+1);
      copy(input+n/2, output+n/2, n-n/2, depth+1);
    }
  }
}

int main() {
        #pragma omp parallel
        #pragma omp single
        copy(X,Y,N,0);
}
```

Problem 4 (4.5 points) Given the following C code excerpt:

```
#define MAXHISTO P // P is the number of cores
typedef struct {
    double total;
    unsigned long long num;
} telem;
telem histo[MAXHISTO];
double v[MAXELEM]; // MAXELEM is a very large value
int i, pos;
 ...
// histo initialization phase
for (i=0; i<MAXHISTO; i++) {
   histo[i].total = 0;
   histo[i].num = 0;
}
// histo computation phase
for (i=0; i<MAXELEM; i++) {
   pos = getpos (v[i], MAXHISTO);   // returns a value between 0 and MAXHISTO-1
   // complex update of field "total" from vector "histo" at position "pos"
   complex_update (&histo[pos].total, v[i]);
   histo[pos].num ++;
}
```

We ask you to:

1. (1.5 points) Add the necessary `OpenMP` pragmas and clauses to parallelize the code in *histo computation* phase, on `P` cores, making use of *implicit tasks* and applying an *INPUT geometric block data decomposition*. The value `MAXELEM` is not necessarily a multiple of `P`. Your solution should maximize parallelism among implicit tasks.

   Solution:

```
omp_lock_t locks[P];
int i, pos;
 ...
// histo initialization phase
for (i=0; i<MAXHISTO; i++) {
   histo[i].total = 0;
   histo[i].num = 0;
   omp_init_lock (&locks[i]);
}
// histo computation phase
#pragma omp parallel private (i, pos) num_threads(P)
{
 int id = omp_get_thread_num();
 int BS = MAXELEM / P;
 int mod = MAXELEM % P;
 int start = id * BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    } else {
       start += mod;
       end += mod;
    }
 }
 for (i=start; i<end; i++) {
```

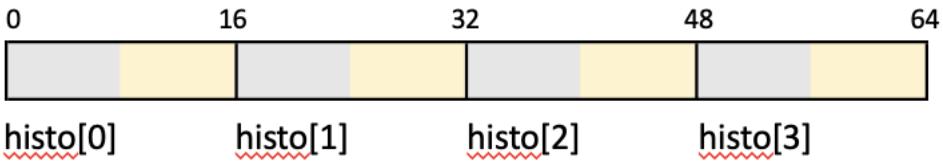
```
    pos = getpos (v[i], MAXHISTO);

    omp_set_lock (&locks[pos]);
    complex_update (&histo[pos].total, v[i]); // complex update of field "total" from vector
    omp_unset_lock (&locks[pos]);

    #pragma omp atomic
    histo [pos].num ++;
  }
 }
 ...
```

2. (1.5 points) Let's consider a shared-memory multiprocessor system composed by two NUMA nodes, each with two processors (cores) and 16 GBytes of shared main memory (MM). Each core has a private cache of 8MBytes. Cache and memory lines are 32 bytes wide. Cores 0 and 1 are allocated in NUMA node 0, and cores 2 and 3 are allocated in NUMA node 1. Data coherence is maintained using Write-Invalidate MSI protocols, with a Snoopy attached to each cache memory to provide coherency within each NUMA node and Write-Invalidate MSU directory-based coherence among the two NUMA nodes.

   (a) Consider `P=4` and draw a picture to show how many memory lines are necessary to allocate vector `histo` in the MM of the previously described system. You need to know that `sizeof(double) = 8 Bytes` and `sizeof(unsigned long long) = 8 Bytes`, the allocation of vectors `v` and `histo` are aligned to the start of memory line.

   Solution:

   To store vector `histo` it is necessary two memory lines (64 bytes):



   (b) **Compute the amount of bits taken by each snoopy** to maintain the coherence between caches inside a NUMA node and, **compute the amount of bits in each node directory** to maintain the coherence among NUMA nodes **ONLY for vector `histo`**.

   Solution:

   To store vector `histo` we need 2 memory lines. In order to keep coherency within a NUMA node using MSI Snoopy protocol, we need to keep 2 bits for each cache line. So the total number of bits needed by the MSI Snoopy protocol is **4 bits**.

   We need two entries in the directory structure to store the needed information to keep coherency. Each entry contain 2 bits for the state (MSU) and 2 bits for the sharers list. Consequently we need $2 * (2 + 2) =$ **8 bits** to keep coherency among NUMA nodes for vector `histo`.

   (c) Assuming that vector `histo` is entirely allocated in the MM of Numa Node 0, and all cache memories of the multiprocessor system are empty, **fill in the attached table** with the sequence of processor commands (Core), bus transactions within NUMA nodes (Snoopy), transactions yes/no between NUMA nodes (Directory), the presence bits, state for each cache and memory line, to keep cache coherence, AFTER the execution of each of the following sequence of commands:

   i. $core_2$ reads the contents of histo[1].total
   ii. $core_2$ writes the contents of histo[2].num
   iii. $core_1$ reads the contents of histo[0].total

3. (1.5 points) Let's assume that we want to execute the program on the shared-memory multiprocessor system described previously. Decide the most appropriate *geometric data decomposition* and write the parallel code in order to minimize synchronizations and reduce coherence traffic. You can also re-write the data structures.

Solution:

We define an OUTPUT geometric block data decomposition on vector `histo`, where the block size is equal to one element. In order to reduce coherence traffic generated by the false sharing when writing to different elements of `histo` but are allocated to the same cache line, we add padding (extra bytes) the the `telem` data structure. No synchronization is needed among implicit tasks as they update different elements of vector `histo`.

```
#define CACHE_LINE_SIZE 32
typedef struct {
    double total;
    unsigned long long num;
    char padding[CACHE_LINE_SIZE - sizeof(double) - sizeof(unsigned long long)];
} telem;
telem histo[MAXHISTO];
 ...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 histo[id]=0;
 for (int i=0; i<MAXELEM; i++)
   int pos = getpos (v[i], MAXHISTO); // returns a value between 0 and MAXHISTO-1
   if (id == pos) {
      histo [pos].total += v[i];
      histo [pos].num ++;
   }
 }
 ...
```

Surname, name .....

Answers to **Question 4, Section 2.c**

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory (yes/no)			cache0	cache1	cache2	cache3
$core_2$ reads histo[1].total									
$core_2$ writes histo[2].num									
$core_1$ reads histo[0].total									

Solution:

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory (y/n)			cache0	cache1	cache2	cache3
$core_2$ reads histo[1].total	$PrRd_2$	$BusRd_2$	y	10	S	-	-	S	-
$core_2$ writes histo[2].num	$PrWr_2$	$BusRdX_2$	y	10	M	-	-	M	-
$core_1$ reads histo[0].total	$PrRd_1$	$BusRd_1$, $Flush_2$	n	11	S	-	S	S	-

PAR – Final Exam Theory/Problems– Course 2023/24-Q1

January 17^{th}, 2024

Problem 1 (6.0 points)

Given the following code fragment written in C:

```
int m[N][N];
...
for (int i=0; i<N; i++) {
   for (int k=0; k<N; k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]); // 100 time units
   }
 }
```

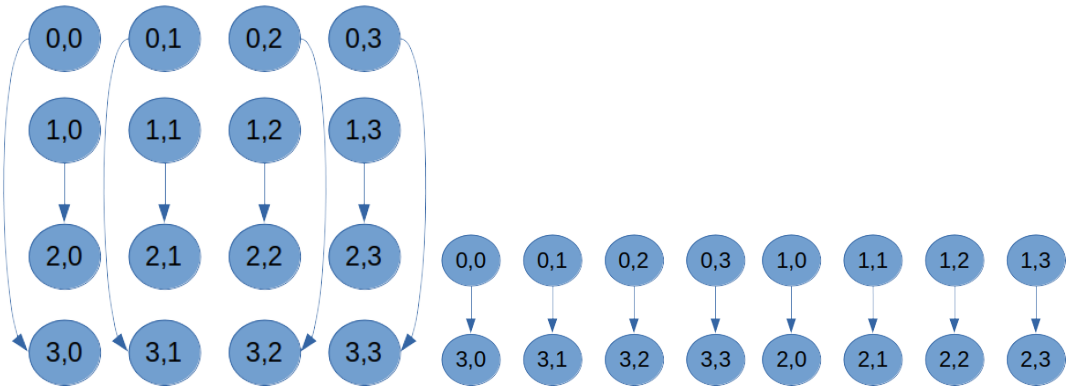
We ask you to:

1. (1.0 Points) Assume 1) a very fine-grain strategy in which a task is defined for each single k loop iteration, with cost equal to 100 t.u. (time units); 2) function `calculation` only reads the two values received as arguments and does not modify any other memory locations; and 3) the rest of variables in the program are stored in registers of the processors. **Also assume $N = 4$ for the following two subquestions:**

   (a) Draw the *Task Dependency Graph (TDG)*, identifying each task in the *TDG* with a label (i, k), being i and k the values of the `i` and `k` loop control variables, respectively.

   Solution:

   Two possible views of the TDG.



   (b) Compute the values for the metrics T_1, T_∞ and P_{min}.

   Solution:

   $T_1 = 16 \times 100 = 1600$ time units.

   $T_\infty = 2 \times 100 = 200$ time units.

   $P_{min} = 8$.

2. (2.5 Points) Write an *OpenMP* parallel version of the code following the most appropriate *geometric data decomposition strategy* for matrix `m` considering that your parallel code should: a) minimize the synchronization overhead among implicit tasks based on the previous *TDG* analysis; 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). Assume that matrix `m` is allocated in memory by rows and that N is a multiple of 2 and not necessarily multiple of the number of processors to be used to execute the program in parallel.

   Solution:

```
int m[N][N];
...
```

```
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = N/nths;
 int mod = N%nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    }
    else {
       start += mod;
       end += mod;
    }
 }
 for (int i=0; i<N; i++) {
   for (int k=start; k<end; k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]);
   }
 }
```

We apply a *block geometric data decomposition by columns*. There are RAW (true) dependencies among elements from the same column, consequently blocks are defined by columns, ensuring no need for synchronization among implicit tasks. The size of the blocks are adjusted as N is not guarantee to be a multiple of the number of threads. In this way, the unbalance is limited to N elements (an entire column).

3. (2.5 Points) Modify the previous *OpenMP* parallel version or write a new one and, if necessary, change the definition of matrix `m` to follow the most appropriate *geometric data decomposition strategy* considering that: a) the program is executed on a parallel machine where memory lines are 128 bytes long; b) matrix `m` is allocated in memory so that its first element is aligned to the start of a memory line and `sizeof(int)` $= 4$ bytes; and c) you can not consider that N is multiple of the size of one element. Your proposed solution should: a) maximize parallelism among implicit tasks; b) maximize data locality and reduce coherence traffic; and c) minimize load unbalance.

Solution:

```
#define MEMLINESIZE 128
#define X MEMLINESIZE/sizeof(int)
int m[N][N + (X - N % X)];
...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = (N/2)/nths;
 int mod = (N/2)%nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
    if (id < mod) {
       start += id;
       end = start + BS + 1;
    }
    else {
```

```
        start += mod;
        end += mod;
     }
  }

  for (int i=start; i<end; i++) {
    for (int k=0; k<N; k++) {
        m[i][k] = calculation (m[N-1-i][k], m[i][k]);
        m[N-1-i][k] = calculation (m[i][k], m[N-1-i][k]);
    }
  }
}
```

In the proposed solution we apply a *block geometric data decomposition strategy by rows* on the first half of the matrix `m`. The rows on the second half are processed by the thread that processes the mirror row in the first half of the matrix, so the code is modified accordingly. In this way dependencies are preserved and there is no need for synchronization between implicit tasks. In order to minimize coherence traffic, we avoid false sharing by adding padding as extra columns at the end of each row of the matrix, so that we complete an entire memory line in case N is not a multiple of the memory line size. In this way we avoid false sharing. Data locality is preserved as the matrix processing is performed by rows. Finally, unbalance is minimized to two rows (2N elements) in the worst case.

Another solution could be to modify the proposed solution of the previous part, and apply a *block cyclic gemotreci data decomposition by columns*:

```
#define MEMLINESIZE 128
#define BS MEMLINESIZE / sizeof(int)
int m[N][N + (BS - (N % BS)];
...
#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();

 for (int i=0; i<N; i++) {
   for (int kk=id*BS; kk<N; kk+=nths*BS) {
      for (int k=kk; k<min(kk+BS,N); k++) {
       m[i][k] = calculation (m[N-1-i][k], m[i][k]);
   }
 }
}
```

Padding is also added to avoid false sharing and consequently minimize coherence traffic. The size of the block ensures data locality. This solution has the disadvantage with respect to the previous proposed one, that the unbalance in the worst case could be up to (BS-1) columns, that is (BS-1)*N elements.

Problem 2 (4.0 points) Assume a NUMA system with two NUMAnodes, each NUMAnode with 16GB of main memory and 2 processors (cores). Each processor has its own local cache memory of 4MB (cache line size of 128 bytes). Data coherence is maintained using *Write-Invalidate MSI* protocols, with a Snoopy attached to each per–core local cache memory to provide coherency within each NUMAnode and directory-based coherence among the two NUMAnodes.

We ask you to:

1. (2.0 Points) Compute the total number of bits that are used by each snoopy to maintain the coherence between caches inside a NUMAnode and, the total number of bits in each node directory to maintain coherence among NUMAnodes.

   Solution:

   Intra-node coherence:

   The number of cache lines is calculated dividing the memory in a cache (4MB) by the cache line size (128B). Therefore: Number of cache lines $= \frac{4MB}{128B} = \frac{4\times 2^{20}}{2^7} = 2^{15}$ entries

   We need 2 bits per entry, so the total number of bits used by each snoopy is: 2^{16}

   Inter-node coherence:

   The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (16GB) by the memoy line size (128B). Therefore:

   Number of Directory Entries $= \frac{16GB}{128B} = 2^{27}$ entries

   Each entry has 4 bits: 2 Presence bits and 2 bits for the Status

   Thus, with 4 bits per directory entry, the total number of bits per NUMAnode is: $4 \times 2^{27} = 2^{29}$

2. (2.0 Points) Let's consider the following definition of matrix `m` with a given value of `N`:

   ```
   #define N 32
   int m[N][N];
   ```

   Assume that a) matrix `m` is allocated in memory aligned to the start of a memory line and b) `sizeof(int)` = 4 bytes.

   The code initializing matrix `m` is executed entirely by $core_0$ (on $NUMAnode_0$) so at the end of the initialization the entire matrix is loaded into the cache of $core_0$. Consequently, all memory lines and cache lines that hold matrix `m` are in *Modified state* with *presence bits = 01* in the directory of the *home* NUMAnode. **We ask you to:** fill in the attached table the sequence of processor commands (Core), cache Miss or Hit, bus transactions within NUMA nodes indicating clearly which attached Snoopy generated the command (Snoopy), state for each cache line affected (for each cache memory of the system, MC_0 to MC_3), state for each memory line (Directory state) and the presence bits, to keep cache coherence, AFTER the execution of each of the following of commands:

   - $core_0$ from $NUMAnode_0$ reads `m[0][0]`
   - $core_0$ from $NUMAnode_0$ reads `m[1][24]`
   - $core_3$ from $NUMAnode_1$ writes `m[0][24]`
   - $core_3$ from $NUMAnode_1$ reads `m[1][24]`
   - $core_0$ from $NUMAnode_0$ writes `m[0][0]`
   - $core_3$ from $NUMAnode_1$ writes `m[1][24]`

Solution:

Action	Core	Cache Miss/Hit	Snoopy	State MC_0	State MC_1	State MC_2	State MC_3	Directory State	Presence bits
$core_0$ reads m[0][0]									
$core_0$ reads m[1][24]									
$core_3$ writes m[0][24]									
$core_3$ reads m[1][24]									
$core_0$ writes m[0][0]									
$core_3$ writes m[1][24]									

Action	Core	Cache Miss/Hit	Snoopy	State MC_0	State MC_1	State MC_2	State MC_3	Directory State	Presence bits
$core_0$ reads m[0][0]	$PrRd_0$	Hit	-	M	-	-	-	M	01
$core_0$ reads m[1][24]	$PrRd_0$	Hit	-	M	-	-	-	M	01
$core_3$ writes m[0][24]	$PrWr_3$	Miss	$BusRdX_3$ $BusRdX$ in $NumaNode_0$ issued by Hub $Flush_0$	I	-	-	M	M	10
$core_3$ reads m[1][24]	$PrRd_3$	Miss	$BusRd_3$ $BusRd$ in $NumaNode_0$ issued by Hub $Flush_0$	S	-	-	S	S	11
$core_0$ writes m[0][0]	$PrWr_0$	Miss	$BusRdX_0$ $BusRdX$ in $NumaNode_1$ issued by Hub $Flush_3$	M	-	-	I	M	01
$core_3$ writes m[1][24]	$PrWr_3$	Hit	$BusUpgr_3$ $BusUpgr$ in $NumaNode_0$ issued by Hub	I	-	-	M	M	10

PAR – Final Exam Theory/Problems– Course 2022/23-Q2

June 21^{th}, 2023

Problem 1 (2.5 points)

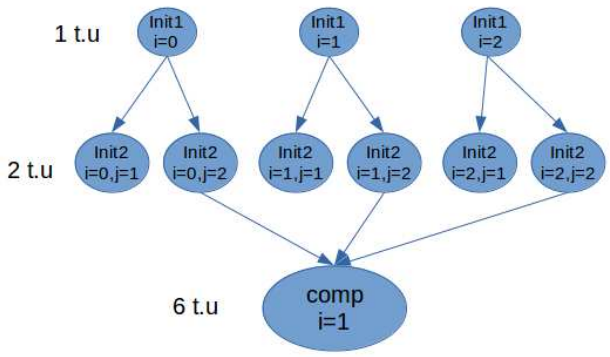
Given the following code including *Tareador* task annotations:

```
#define N 3
int I[N][N];
int R[N][N];
...
for (int i=0; i<N; i++) {
    tareador_start_task ("init1");
    I[i][0] = foo(i);   // cost = 1 t.u.
    tareador_end_task ("init1");

    for (int j=1; j<N; j++) {
       tareador_start_task ("init2");
       I[i][j] = j*I[i][0]; // cost = 2 t.u.
       tareador_end_task ("init2");
} }

for (int i=1; i<N-1; i++) {
    tareador_start_task ("comp");
    for (int j=i+1; j<N; j++)
       R[i][j] = i*(I[i-1][j]+I[i][j]+I[i+1][j]); // cost = 6 t.u
    tareador_end_task ("comp");
}
```

Assume: a) the execution time for tasks `init1` and `init2` is 1 and 2 t.u., respectively; b) the execution time for each iteration of the loop body for task `comp` is 6 t.u.; c) the rest of code has negligible cost; d) all scalar variables (i.e. all variables except matrices `I` and `R`) are stored in registers; e) function `foo` does not access any memory location during its execution; and f) `N` with the value defined in the code above. **We ask you** to:

1. Draw the Task Dependence Graph (*TDG*), indicating the cost of each task. Label each task with the values of i, and when necessary, with the pair of values i and j.

   Solution:



2. Based on the *TDG*, compute the values for T_1 and T_∞.

   Solution:

   $T_1 = 1 \times 3 + 2 \times 6 + 6 \times 1 = 21$ time units.

   $T_\infty = 1 + 2 + 6 = 9$ time units , determined by the critical path: *init1, init2, comp.*

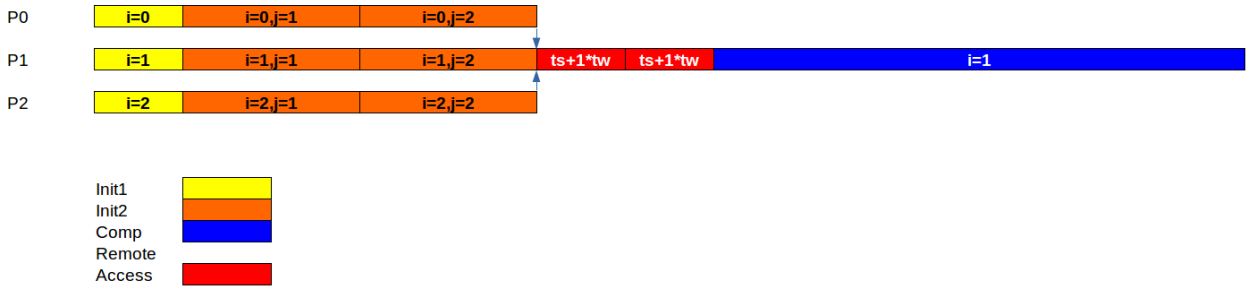
Next consider the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses do not introduce any overhead, but an access to data in different processors introduces a data–sharing overhead, determined by $t_{comm} = t_s + m \times t_w$; t_s is the "start-up" time, t_w is transfer time for one element and m the number of elements transferred during the remote access. Also, according to the data sharing model: at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Assuming a) the number of processors p is equal to N; b) matrices `I` and `R` are already distributed in the memory of each processor when the computation starts, following a *block row data decomposition*; c) the resulting matrices should remain distributed in the same way at the end of the execution; and d) tasks are assigned to processors following the *owner-computes rule* so that a task is executed by the processor that owns the memory that holds the output of that task. **We ask you:**

3. Draw the timeline for the execution with $p = 3$, showing both the computational and remote memory access costs.

   Solution:

   Block row data decomposition of a matrix with a number of rows N equal to P means that each processor i will have row i of the matrix. Therefore, tasks `init1` and `init2` accessing row i of matrix I are assigned to processor i because their own row i of that matrix. Task `comp` access row 1 of matrix R. Then, it is assigned to processor 1. Following owner-compute rules, the mapping and execution of the tasks follows:



4. Obtain the expression that determines the parallel execution time on $p = 3$ processors (T_p), as a function of t_s and t_w, assuming the task durations obtained before.

   Solution:

   $T_p = 1 + 2 \times 2 + 2 \times (t_s + 1 \times t_w) + 6 = 11 + 2 \times (t_s + 1 \times t_w)$ time units

Problem 2 (2.5 points)

Given the following recursive sequential code:

```
#define N_MAX (1<<29) // 512*1024*1024
struct t_person{
   t_data data; // personal information of bank client
   float balance; // current balance for client
   float interest; // interest for client
};
struct t_person best_client;

void find_best_client(struct t_person *people, int n) {
   int n2 = n/2;
   if (n==1) {
      if (best_client.balance < people[0].balance)
          best_client = people[0]; // copy all info of the person to best_client
   } else {
       find_best_client(people, n2);
       find_best_client(people+n2, n-n2);
} }

int main() {
   struct t_person bank_info[N_MAX];
   ...
   best_client.balance=0.0;
   find_best_client(bank_info, N_MAX);
   ...
}
```

Write an OpenMP version following a tree recursive task decomposition, taking into account the task creation overheads and minimizing both task synchronizations and synchronizations to update shared variables.

A Solution:

```
... // same declarations

#define CUTOFF (4)

void find_best_client(struct t_person *people, int n, int depth) {
   int n2 = n/2;
   if (n==1) {
      if (best_client.balance < people[0].balance) // FIRST TEST (reduce # criticals)
      #pragma omp critical
      {
        if (best_client.balance < people[0].balance) // SECOND TEST (necessary!)
          best_client = people[0]; // copy all info of the person to best_client
      }
   } else {
      if (omp_in_final())
      {
       find_best_client(people, n2, depth);
       find_best_client(people+n2, n-n2, depth);
      } else {
       #pragma omp task final(depth>=CUTOFF)
       find_best_client(people, n2, depth+1);
       #pragma omp task final(depth>=CUTOFF)
       find_best_client(people+n2, n-n2, depth+1);
       // NO TASKWAIT
      }
} }
```

```
int main() {
   struct t_person bank_info[N_MAX];
   ...
   best_client.balance=0.0;
   #pragma omp parallel
   #pragma omp single
   find_best_client(bank_info, N_MAX, 0);
   ...
}
```

Problem 3 (2.5 points)

Given the following sequential C code:

```
#define N 10000000
#define NUMBINS 100
int input[N];
int histogram[NUMBINS];

int findMax(int *v); // returns the maximum value encountered in vector v
int findMin(int *v); // returns the minimum value encountered in vector v
...
int min = findMin(input);
int max = findMax(input);
int binInterval = (max-min)/NUMBINS;

for (int i=0; i<N; i++) {
    int bin = (input[i]-min)/binInterval;
    histogram[bin]++;
}
```

We ask you to:

1. Implement a parallel *OpenMP* version of the code that follows an ***output block geometric data decomposition***, where synchronization overheads are minimized and parallelism is maximized.

   Solution:

```
#define N 10000000
#define NUMBINS 100
int input[N];
int histogram[NUMBINS];

int findMax(int *v); // returns the maximum value encountered in vector v
int findMin(int *v); // returns the minimum value encountered in vector v
...
int min = findMin(input);
int max = findMax(input);
int binInterval = (max-min)/NUMBINS;

#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = NUMBINS / nths;
 int mod = NUMBINS % nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
   if (id < mod) {
      start += id;
```

```
      end = start + BS + 1;
    }
    else {
      start += mod;
      end += mod;
    }
  }
  for (int i=0; i<N; i++) {
     int bin = (input[i]-min)/binInterval;
     if (bin >= start && bin < end)
         histogram[bin]++;
  }
 }
```

2. Assume that the processor's cache line size is 64 bytes, also that the size of the `int` data type is 4 bytes and the initial addresses of `input` and `histogram` vectors are both aligned with the start of a cache line. Write a new parallel *OpenMP* version of the code (without changing the definition of the used data structures) to avoid *false sharing* overheads.

   Solution:

   The cache line size is 64 bytes, so the total number of elements from `histogram` that fit in one cache line is 16. In order to avoid false sharing we propose a block-cyclic data decomposition in such a way that each block has a number of elements that fit in one cache line, so each thread should be allocated a block of size 16.

```
#define N 10000000
#define NUMBINS 100
#define CACHE_LINE_SIZE 64
int input[N];
int histogram[NUMBINS];

int findMax(int *v); // returns the maximum value encountered in vector v
int findMin(int *v); // returns the minimum value encountered in vector v
...
int min = findMin(input);
int max = findMax(input);
int binInterval = (max-min)/NUMBINS;

#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = CACHE_LINE_SIZE / sizeof(int);

 for (int i=0; i<N; i++) {
    int bin = (input[i]-min)/binInterval;
    if ((bin/BS)%nths == id)
        histogram[bin]++;
 }
}
```

Problem 4 (2.5 points) Assume a multiprocessor system composed of two NUMA nodes, each with 16 GB of main memory. Each NUMA node has two cores (NUMAnode0: core0-1, NUMAnode1: core2-3), each core with a private cache of 4 MB. Cache and memory lines are 32 bytes wide and data coherence in the system is maintained using Write-Invalidate protocols, with a Snoopy attached to each cache memory to provide MSI coherency within each NUMA node and a MSU directory-based coherence among the two NUMA nodes.

Given the following parallel *OpenMP* code excerpt to be executed on the described system:

```
1   #define IMAGESIZE 128*128
2   #define MAXITERS 100
3   typedef float tpixel[3]; // r,g,b values
4   tpixel image[IMAGESIZE], result[IMAGESIZE];
    ...
5   initialization (result); // on core 0
6   #pragma omp parallel num_threads(3)
7   {
8     int id = omp_get_thread_num();
9     for (int iter=0; iter<MAXITERS; iter++)
10        for (int i=0; i<IMAGESIZE; i++)
11           result[i][id] = apply_func (image[i][id],   // apply_func does not affect
12                                       result[i][id]); // any other variables
13  }
```

and assuming that: 1) vectors `image` and `result` are the only variables that will be stored in memory (the rest of variables will be all in registers of the processors); 2) the initial addresses of `image` and `result` vectors are aligned with the start of a cache line; 3) the size of a `float` data type is 4 bytes; and 4) $thread_i$ always executes on $core_i$, where i = [0,1,2], **we ask you to:**

1. Indicate how many entries in the directory structure are needed to store the coherence information of the `result` vector. In addition, for each entry indicate the number of bits needed and their function.

   Solution:

   Each element of the `result` vector has 12 bytes of total size ($3 \times 4bytes = 12bytes$), the total size of the vector would be: $12 \times (128 \times 128) = 12 \times 2^{14}$

   The number of memory lines needed to allocate the `result` vector is : $(12 \times 2^{14})/2^6 = 12 \times 2^8$

   For each memory line it is necessary to keep the status information (*M,S,U*), which can be stored in 2 bits and the presence bits (one bit per NUMA node, total = 2 bits).

   The total number of bits needed is : $12 \times 2^8 \times (2 + 2)$

2. Compute the total number of bits that are required to maintain the coherence information at cache level within each NUMA node. Indicate also the number of bits per cache line and their function.

   Solution:

   Each cache has a total size of 4 MB and the cache line size is 64 bytes, so the total number of lines per cache = $2^{22}/2^6 = 2^{16}$ cache lines per cache.

   For each cache line it is necessary to keep the status information (*M,S,I*), which can be stored in 2 bits.

   So finally, the total number of bits necessary to keep the information of the Snoopy MSI coherence protocol is: $2^{16}2 = 2^{17}$

3. Complete the table provided with information for each enumerated memory operation (after it is completed), from the previous code, executed by the threads in the parallel region. You should specify the relative number of the memory line affected (with respect to the start address of the *result* vector, i.e. *result[0][0]* affects line 0), hit or miss, bus transactions of the MSI Snoopy protocol, state of the cache line in the MSI Snoopy protocol, whether there are or not commands from the Directory protocol between NUMA nodes (yes or no), state of the memory line in the Directory and, Presence bits.
4. The speedup achieved when executing the previous parallel code, with three threads, is far below 3x. Explain briefly the reason and suggest any modification to the data structures in order to improve its performance. Re-write the necessary code.

Action	Memory line	Cache Miss/Hit	Bus command	State MC_0	State MC_1	State MC_2	NUMA comands (y/n)	Directory State	Presence bits
Initial state result[0][0:2]				M	-	-		M	01
$thread_1$ reads result[0][1]									
$thread_2$ reads result[0][2]									
$thread_1$ writes result[0][1]									
$thread_0$ reads result[0][0]									

Solution									
Action	Memory line	Cache Miss/Hit	Bus command	State MC_0	State MC_1	State MC_2	NUMA comands (y/n)	Directory State	Presence bits
Initial state result[0][0:2]				M	-	-		M	01
$thread_1$ reads result[0][1]	0	Miss	$BusRd_1$/ $Flush_0$	S	S	.	n	S	01
$thread_2$ reads result[0][2]	0	Miss	$BusRd_2$	S	S	S	y	S	11
$thread_1$ writes result[0][1]	0	Hit	$BusUpgr_1$	I	M	I	y	M	01
$thread_0$ reads result[0][0]	0	Miss	$BusRd_0$/ $Flush_1$	S	S	I	n	S	01

Solution:

Each element of the `result` vector has 3 fields of `float` data type, computing a total size of 12 bytes per element. The size of a cache line is 64 bytes. Given that each element of the result vector is being updated by the three threads, a false sharing situation occurs with high probability (every time more than one thread is updating the same entry at the result vector).

In order to avoid such inefficiency we can apply "padding", that is add some extra bytes to prevent different threads from accessing the same cache line for updating it concurrently. One possible solution could be to add extra bytes following each field of the `tpixel` structure in order to complete a cache line (number of extra bytes = 64 - 4 = 60 bytes, which makes 60/4 = 15 extra elements) In this case we need to substitute line 3 of the code by:

```
1   #define IMAGESIZE 128*128
2   #define MAXITERS 100
3   typedef float tpixel[3][1+15]; // r ,g ,b values with padding
4   tpixel image[IMAGESIZE], result[IMAGESIZE];
    ...
5   initialization (result); // on core 0
6   #pragma omp parallel num_threads(3)
7   {
8     int id = omp_get_thread_num();
9     for (int iter=0; iter<MAXITERS; iter++)
10        for (int i=0; i<IMAGESIZE; i++)
11           result[i][id][0] = apply_func (image[i][id][0],   // apply_func does not affect
```

```
12                                      result[i][id][0]); // any other variables
13   }
```

PAR – Final Exam Laboratory– Course 2022/23-Q2

June 21^{th}, 2023

Problem 1: Lab 1 (2.5 points)

Given the following outputs obtained after the interactive execution of `pi_omp` with different number of threads:

```
par@boada-7> OMP_NUM_THREADS=1 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 1 threads = 3.141592653589828
Execution time (secs.): 2.368062
2.36user 0.01system 0:02.37elapsed 99%CPU (0avgtext+0avgdata 4320maxresident)k
0inputs+0outputs (1major+275minor)pagefaults 0swaps

par@boada-7> OMP_NUM_THREADS=2 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 2 threads = 3.141592653589845
Execution time (secs.): 1.186354
2.36user 0.00system 0:01.19elapsed 198%CPU (0avgtext+0avgdata 4668maxresident)k
0inputs+0outputs (1major+381minor)pagefaults 0swaps

par@boada-7> OMP_NUM_THREADS=4 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 4 threads = 3.141592653589845
Execution time (secs.): 1.188191
2.36user 0.01system 0:01.19elapsed 198%CPU (0avgtext+0avgdata 4672maxresident)k
0inputs+8outputs (1major+306minor)pagefaults 0swaps
```

1. Explain the meaning the four first values reported by the `/usr/bin/time` command when executed with a single thread: *2.36user 0.01system 0:02.37elapsed 99%CPU*.

   Solution:

   They are the user CPU time, system CPU time, elapsed time and % of elapsed time dedicated to CPU time (user+system CPU time*100/elapsed time), respectively.

2. When executed with 2 and 4 threads, explain why *2.36user* does not change with respect to the value reported for a single thread, and why the *198%CPU* is the same for 2 and 4 threads.

   Solution:

   The value *2.36user* corresponds to the addition of the CPU time of each of the 2 or 4 threads during the execution. Each thread lasts of $2.36/nt$ approx, being nt the number of threads.

   The %CPU is close to 200% for 2 threads because the CPU time remains the same while the elapsed time is half the elapsed time of a single thread execution. In the case of 4 threads is still the same because we only have two cores in interactive mode, and the elapsed time remains the same than the case with 2 threads.

Problem 2: Lab 3 (2.5 points)

Assuming the following task decomposition strategies to parallelize the *Mandelbrot set*, which one do you think would be more efficient? Please reason your answer taking into account the number of tasks generated/executed and their granularity for each strategy.

```
#pragma omp parallel    // Code 1
#pragma omp single
 for (int row = 0; row < height; ++row)
   for (int col = 0; col < width; ++col) {
     #pragma omp task firstprivate(row,col)
     {
             . . .
   } }
```

```
#pragma omp parallel    // Code 2
#pragma omp single
 #pragma omp taskloop
 for (int row = 0; row < height; ++row) {
   for (int col = 0; col < width; ++col) {
             . . .
 } }
```

Solution:

Code 2 is much more efficient than Code 1. Code 1 corresponds with the first point strategy implementation that you evaluated in the laboratory session. One of the threads in Code 1 has to create a huge number of very fine-grain tasks sequentially. This incurs a significant overhead of task creation. Code 2 corresponds with the row strategy implementation with taskloop that you also evaluated in the laboratory session. This reduces the number of tasks created, which usually is proportional to the number of threads in the parallel region and much less than the number of tasks created in Code 1. The task creation overhead is significantly reduced with larger task granularities.

Problem 3: Lab 4 (2.5 points)

Consider the following sequential *multisort* code.

```
 1   void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
 2    if (length < MIN_MERGE_SIZE*2L) { // Base case
 3       basicmerge(n, left, right, result, start, length);
 4    } else { // Recursive decomposition
 5       merge(n, left, right, result, start, length/2);
 6       merge(n, left, right, result, start + length/2, length/2);
 7   }}
 8
 9  void multisort(long n, T data[n], T tmp[n]) {
10     if (n >= MIN_SORT_SIZE*4L) {     // Recursive decomposition
11        multisort(n/4L, &data[0], &tmp[0]);
12        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
13        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
14        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
15
16        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
17        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
18
19        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
20    } else     // Base case
21        basicsort(n, data);
22  }
23  void main() {
24   ...
25   multisort(n,elements,elements_tmp);
36   ...
37  }
```

Let's suppose that you parallelize it following a tree strategy decomposition with a given cut-off level. Assuming the laboratory problem size, 1024K elements to sort, and MIN_SORT_SIZE and MIN_MERGE_SIZE equal to 256. Answer the following questions:

1. Assume you want to use the `final(depth>=cut_off)` clause in the task pragmas, being "depth" the recursion level. Indicate where (i.e, in the sequential code line number above) you will add the task creation control to continue creating tasks or not, and write the code line to do it.

   Solution:

   Substitute the code at lines 5-6 by the following code structure:

   ```
   11  if (!omp_in_final())
       {
         Code with tasks
       } else {
         Code without tasks
       }
   ```

Also, substitute the code at lines 11-19 with the same code structure.

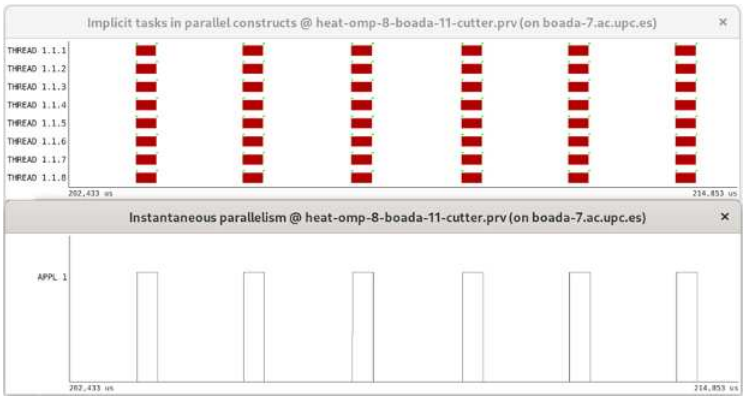
2. Assume now you are using task with depend clauses (with no cut-off control). Indicate the synchronisation (`depends` clauses, `taskwaits` or/and `taskgroups`) that you need to add in your tree-recursive task decomposition (i.e the sequential code line number, before and/or after you will insert them) when using tasks with depend clauses to guarantee the functionality of the *multisort* program.

   Solution:

   No solution provided. Look at your laboratory deliverable and feedback.

Problem 4: Lab 5 (2.5 points)

Assume that you have parallelized *Jacobi* solver function with the data decomposition seen at the laboratory. You run *modelfactors* and use the instantaneous parallelism and implicit tasks in parallel constructs hints with *Paraver* to figure out why *Jacobi* application is not scaling properly. The "zoom in capture" of part of the execution trace for 8 threads of the first implementation version of heat application with *Jacobi* solver follows (top: implicit tasks in parallel constructions, bottom: instantaneous parallelism):



We ask you to briefly explain your answer to the following questions:

1. What do you observe with instantaneous parallelism hint at the view of *Paraver*? What is the instantaneous parallelism achieved during the *Jacobi* solver function and during the execution of the other parts of the code?

   Solution: The amount of useful work done (state running) by all the threads at a certain instant of time. That is, if two threads are running, instantaneous parallelism is 2. The instantaneous parallelism achieved by *Jacobi* is about *nt*, being *nt* the number of threads. However, it is 1 for other parts of the code because they are not parallelized.

2. How did you solve the lack of parallelism and performance problem of the initial *Jacobi* implementation to achieve linear scalability?

   Solution:

   No solution provided. Look at your laboratory deliverable and feedback.

PAR – Final Exam – Course 2022/23-Q1

January 18^{th}, 2023

Problem 1 (2.5 points)

Given the following code with *tareador* task annotations:

```
#define p ...   // Number of processors
#define NR ...  // Number of rows
#define NC ...  // Number of columns

int M[NR][NC];
int BS = NR/p;  // Assume p divides NR exactly

for (int ii=0; ii<NR; ii+=BS) {                    // Matrix initialization
    tareador_start_task ("init");
    for (int i=ii; i<ii+BS; i++)
       for (int j=0; j<NC; j++)
         M[i][j] = init(i,j);                      // <-- Cost ti
    tareador_end_task ("init");
}

tareador_start_task ("comp1");                     // Begin computations
for (int i=0; i<BS; i++)
    for (int j=0; j<NC; j++)
        M[i][j] = comp1(M[i][j]);                  // <-- Cost tb
tareador_end_task ("comp1");

for (int ii=BS; ii<NR; ii+=BS) {                   // Final computations
    tareador_start_task ("comp2");
    for (int i=ii, int i0=0;
         i<ii+BS;
         i++, i0++)
       for (int j=0; j<NC; j++)
         M[i][j] = comp2(M[i0][j], M[i][j]);       // <-- Cost tf
    tareador_end_task ("comp2");
}
```

Let us assume the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses have no cost, but an access to data in different processors introduces a data–sharing overhead: the access time to remote data is determined by $t_{comm} = t_s + m \times t_w$, being t_s and t_w the "start-up" and sending time of an element, respectively, and being m the size of the message. Also, according to the data sharing model: at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Assume that the number of processors p divides the number of rows `NR` exactly; routines `init`, `comp1` and `comp2` do not modify any memory position; the execution time of one iteration of the body of the most internal loops is t_i, t_b and t_f respectively; tasks are scheduled to processes following the *owner-computes rule* so that a task will be executed by the processor who owns the memory that holds the output of that task; the matrix is already distributed in the memory of each processor when the computation starts; the resulting matrix remains distributed and there is no final communication to a single processor; the strategy used for decomposing matrix `M` follows a *block row distribution*: the matrix `M` is distributed so that each processor has $\frac{NR}{p}$ consecutive rows. **We ask** you to:

1. Draw a Task Dependence Graph (TDG) for the case where $p = 4$;
2. Draw a timeline for the execution with $p = 4$;
3. Provide a general expression that determines the parallel execution time on p processors (T_p): express T_p as a function of p, `NR`, `NC`, t_i, t_b, t_f, t_s and t_w.

Solution:

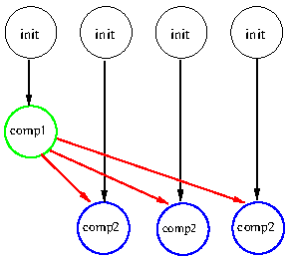


Figure 1: TDG

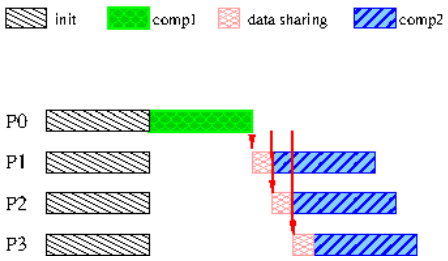


Figure 2: Timeline

$T_p = t_{data\_sharing} + t_{calc}$

$t_{data\_sharing} = (t_s + \frac{NR}{P} \times NC \times t_w) \times (P-1)$

$t_{calc} = \frac{NR}{P} \times NC \times t_i + \frac{NR}{P} \times NC \times t_b + \frac{NR}{P} \times NC \times t_f = \frac{NR}{P} \times NC \times (t_i + t_b + t_f)$

Problem 2 (2.5 points)

Assume a NUMA (non-uniform memory architecture) multiprocessor system composed of 4 NUMA-nodes ($node_{0:3}$), each node with 8 GBytes of main memory and 4 cores ($core_{0:3}$). Each core has only one level of cache memory of 2 MBytes (with cache lines of 32Bytes). The system includes all the necessary mechanisms (seen in class) to keep the memory coherence inside a NUMA-node and between NUMA-nodes.

1. (0.5 points) In order to support a write-invalidate MSI cache-coherence protocol within each NUMA-node, how many additional bits should be used in each cache line, and what are their role and possible values? How many bits in total per NUMA-node (only within each NUMA-node) are used for that purpose?

   Solution:

   We need 2 additional bits per cache line in order to store the state of the cache line. Those 2 bits can code up to four possible states:

   - I invalid
   - S shared (two or more nodes may have clean copies)
   - M modified (dirty)
   - Note that a fourth state can be coded with 2 bits. However, it is not necessary for the MSI protocol.

   The number of cache lines is calculated dividing the memory in a cache (2MB) by the cache line size (32B). Therefore: Number of cache lines $= \frac{2MB}{32B} = 2^{16} = 64K$ entries

   We need 2 bits per entry. And there are a total of 4 cache memories per NUMAnode.

   Thus, the total number of bits amounts to:

   $\frac{2MB}{32B}$ entries/cache $\times 2$ bits/entry $\times 4$ caches/NUMAnode $= 2^{19}$ bits $= 512K$ bits/NUMAnode

2. (0.5 points) In order to support a directory-based write-invalidate MSU cache-coherence protocol between NUMA-nodes, how many bits should be used in the directory for each line in main memory, and what are their role and possible values? How many bits in total per NUMA-node (directory) are used for that purpose?

   Solution:

   The home NUMA-node is in charge of the coherence of its physical memory lines by means of the directory entries. A directory entry stores the line state and the identities of other NUMA-nodes sharing this memory line.

Bits per directory entry:

- Presence bits (nodes currently having the line), 1 bit per NUMA-node: i.e. 4 bits
- State bits (to track the state of memory lines): 2 bits
  Those 2 bits can code up to four possible states:
  - U uncached, not valid in any cache
  - S shared (two or more nodes may have clean copies)
  - M modified (dirty)
  - Note that a fourth state can be coded with 2 bits. However, it is not necessary for the MSI protocol.

Total: 6 bits per directory entry.

The number of entries in the directory structure per NUMA-node is calculated dividing the memory in a NUMA-node (8GB) by the memoy line size (32B). Therefore:

Number of Directory Entries $= \frac{8GB}{32B} = 2^{28} = 256$ Mega entries

Thus, with 6 bits per directory entry, the total number of bits per NUMAnode will be:

$\frac{8GB}{32B}$ entries/NUMAnode $\times 6$ bits/entry $= 3 \times 2^{29} = 1536$ Mega bits/NUMAnode

3. (1.5 points) Consider the following parallel program executed on only two cores (processors) inside $node_0$ of the previous multiprocessor system:

```
...
double A[M][N];
...
```

Core 0: Update even-numbered rows

```
for ( j = 0 ; j < M ; j += 2 )
   for ( k = 0 ; k < N ; k++ )
        A[j][k] = f(j,k);
```

Core 1: Update odd-numbered rows

```
for ( j = 1 ; j < M ; j += 2 )
   for ( k = 0 ; k < N ; k++ )
       A[j][k] = g(j,k);
```

Assume that matrix `A` is stored in main memory of $node_0$ (without copies of any of its elements in the caches of the NUMAnodes), that each element of matrix `A` is 8 Bytes and that the first element of `A` is aligned on a cache line boundary, $N = 2$ and M is a value multiple of 2.

(a) What would be the maximum number of invalidations that would be sent through the bus expressed as a function of M? What is causing such a memory coherence problem?

Solution:

For $N = 2$, every two rows fall in the same cache line, causing up to 3 invalidations due to false sharing for every pair of lines. Thus, the number of invalidations is $3 \times M/2$.

(b) Modify the declaration of matrix A so that you do not have to change the code and avoid the previous coherence problem?

Solution:

We add enough padding to the second dimension of matrix A to make the size (in bytes) of a row of A equal or multiple of the number of bytes of a cache line.

```
...
#define PADDING ((CACHE_LINE_SIZE - (N*sizeof(double) % CACHE_LINE_SIZE))/sizeof(double))
double A[M][N+PADDING];
...
```

Problem 3 (2.5 points)
Given the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   index[tmp]++;
 } }

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...  // Here we have initialized S to random numbers and index to 0's
  histogram(S, N, index);
  ...
}
```

Write two different OpenMP parallel implementations of function `histogram` using the strategies presented below. You can modify the sequential code, add local variables and use any OpenMP pragma (except `omp for` worksharing-loop construct) and function you may need. Both implementations should minimize the use of synchronizations and load unbalance between threads during the processing of vector S.

1. (1 point) Cyclic Data Decomposition of the Output vector `index`.
   Solution:

   ```
   #define SIZE_INDEX 256
   #define N 1024*1024*1024
   ...

   void histogram(unsigned int *S, int n, unsigned int *index)
   {
    unsigned int i, tmp;

   #pragma omp parallel private(i, tmp)
   {
      int id = omp_get_thread_num();
      int num_threads = omp_get_num_threads();

      for (i=0;i<n;i++) {
        tmp = S[i]%SIZE_INDEX;
        if ((tmp%num_threads)==myid)
          index[tmp]++;
    }
   }

   }

   ...
   ```

2. (1.5 points) Block Data Decomposition of the Input vector `S`.
   Solution:

   ```
   #define SIZE_INDEX 256
   ```

```
#define N 1024*1024*1024
...

void histogram(unsigned int *S, int n, unsigned int *index)
{
 unsigned int i, tmp;

 #pragma omp parallel private(i, tmp)
 {
  int myid        = omp_get_thread_num();
  int num_threads = omp_get_num_threads();
  int i_start     = myid * (N/num_threads);
  int i_end       = (myid+1) * (N/num_threads);
  int res         = N%num_threads;
  if (res)
  {
     i_start = i_start + (myid<res)?myid:res);
     i_end   = i_end   +   (myid<res)?myid+1:res);
  }

  unsigned int local_index[SIZE_INDEX];
  for (i=0;i<SIZE_INDEX;i++)
   local_index[i]=0;

  for (i=i_start;i<i_end;i++) {
     tmp = S[i]%SIZE_INDEX;
     local_index[tmp]++;
  }

  for (i=0;i<SIZE_INDEX;i++)
  {
   #pragma omp atomic
   index[i] += local_index[i];
  }

 }
}
```

Problem 4 (2.5 points)

We ask you to write two additional parallel OpenMP implementations of the code that computes the `histogram`, this time using *task decomposition* strategies:

1. (1 point) Write an efficient OpenMP parallel version of the histogram program in Problem 3 using *explicit tasks*.

   Solution: A possible implementation:

```
...
void histogram(unsigned int *S, int n, unsigned int *index)
{
 unsigned int i, tmp;

 #pragma omp parallel
 #pragma omp single
 #pragma omp taskloop private(tmp)
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   #pragma omp atomic
   index[tmp]++;
 }
}
```

2. (1.5 points) Write a *recursive tree task decompostion* with *cutoff* based on the depth of the recursivity for the following code:

```
#define SIZE_INDEX 256
#define N 1024*1024*1024
#define BASE_SIZE 512
#define CUTOFF 3

void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   index[tmp]++;
} }

void histogram_rec(unsigned int *S, int n, unsigned int *index) {
  unsigned int n2=n/2;

  if (n > BASE_SIZE) {
    histogram_rec(S, n2, index);
    histogram_rec(&S[n2], n-n2, index);
  } else {
      histogram(S, n, index);
 }
}

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...
  // Here we have initialized S to random numbers and index to 0's
  histogram_rec(S, N, index);
  ...
}
```

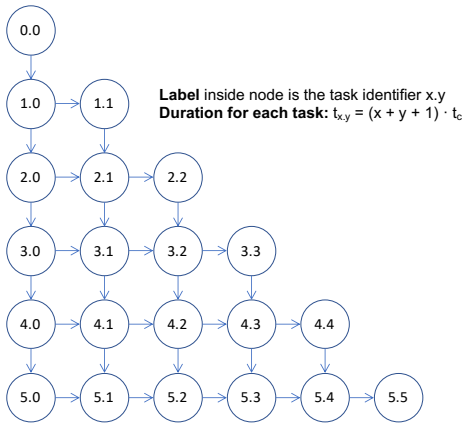
Solution: A possible implementation:

```
...
void histogram(unsigned int *S, int n, unsigned int *index) {
 unsigned int i, tmp;
 for (i=0; i<n; i++) {
   tmp = S[i]%SIZE_INDEX;
   #pragma omp atomic
   index[tmp]++;
 }
}

void histogram_rec(unsigned int *S, int n, unsigned int *index, int depth) {
  unsigned int i, tmp, n2;

  if (n > BASE_SIZE) {
     n2 = n/2;
     if ( !omp_in_final() ) {
       #pragma omp task final(depth>=CUTOFF)
       histogram_rec(S, n2, index, depth+1);
       #pragma omp task final(depth>=CUTOFF)
       histogram_rec(&S[n2], n-n2, index, depth+1);
     }
     else {
       histogram_rec(S, n2, index, depth+1);
       histogram_rec(&S[n2], n-n2, index, depth+1);
     }
  else {
      histogram(S, n, index);
 }
}

void main() {
  unsigned int S[N];
  unsigned int index[SIZE_INDEX];
  ...
  // Here we have initialized S to random numbers and index to 0's
  #pragma omp parallel
  #pragma omp single
  histogram_rec(S, N, index, 0);
  ...
}
```

PAR – Final Exam – Course 2021/22-Q2

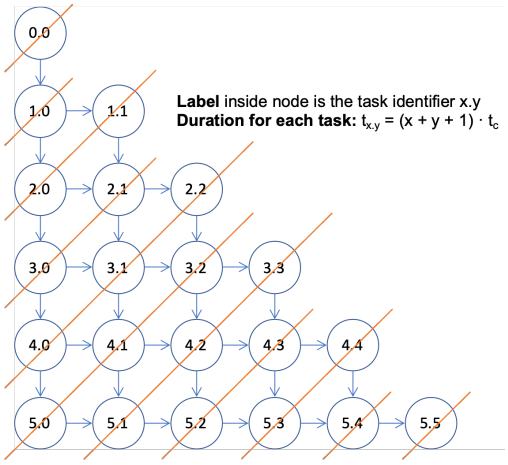
June 20^{th}, 2022

Problem 1 (1.5 points) Given the following Task Dependence Graph associated to the task decomposition applied to a certain program:

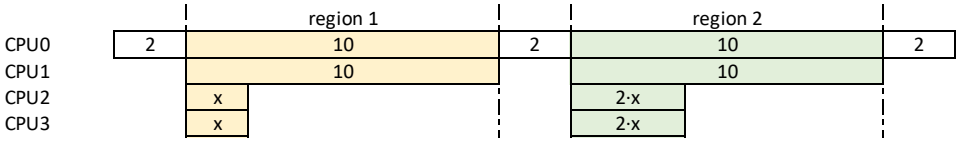


The duration, in time units, of each `compute_task` depends on the value of the row (`x`) and column (`y`) where it is placed (label `x.y`): $t_{x.y} = (x + y + 1) \times t_c$. **We ask you** to compute the values for T_1, T_∞, $Parallelism$ and P_{min}.

Solution: $T_1 = 126 \times t_c$ (sum of the cost of all nodes in the TDG above), $T_\infty = 66 \times t_c$ (defined by the critical path in the TDG, including all nodes in the column on the left and all nodes in the row at the bottom of the TDG above) and $Parallelism = T_1 \div T_\infty = 126 \div 66 = 1.\widehat{90}$. The minimum number of processors to achieve it is $P_{min} = 3$, since we need enough simultaneous processors to execute the widest anti–diagonal in the wavefront execution of the TDG, as shown in the picture below.



Problem 2 (1.5 points) The following diagram plots the timeline for the execution of a parallel application, with two parallel regions, on 4 processors (time evolves from left to right):



Each box represents a burst executed on a processor and the number inside its execution time. There are 3 sequential regions with constant execution time of 2 time units. The execution time is unknown for two of the bursts in $region1$ and in $region2$, both affected by the value x (the timeline shows the case for which

$(2 \times x) < 10$).

- Obtain all the possible values for value x that would lead to an speed-up $S_4 = 2.5$ when the application is executed on 4 processors. Observe that for values $0 < x \leq 5$, $5 < x \leq 10$ and $x > 10$ the two parallel regions have a different contribution in the timeline.

  Solution: In all cases, $T_1 = 2 + (10 + 10 + x + x) + 2 + (10 + 10 + 2 \times x + 2 \times x) + 2 = 46 + 6 \times x$.

  - For x smaller than 5, the execution time for both parallel regions is 10 units, so $T_4 = 2 + 10 + 2 + 10 + 2 = 26$. Therefore from $S_4 = T_1/T_4 = 2.5$ we get $x = 19/6 = 3.16$.
  - For x larger than 5 but smaller than 10, the execution time for the first parallel region is still 10 but for the second one is $2 \times x$. Therefore $T_4 = 16 + 2 \times x$. From $S_4 = T_1/T_4 = 2.5$ it is not possible to get a positive value for x.
  - Finally, for x larger than 10, both parallel regions are dominated by the value of x, being the execution time $T_4 = 6 + 3 \times x$. Again from $S_4 = T_1/T_4 = 2.5$ we obtain $x = 31/1.5 = 20.66$.

- For the values of x in the previous question, obtain the value for $S_{p \to \infty}$, assuming that parallel regions ideally scale with the number of processors.

  Solution: For the two cases above, $T_{p \to \infty} = 6$. Therefore:

  - For $x = 19/6 = 3.16$ we get $S_{p \to \infty} = 65/6 = 10.83$.
  - Similarly, for $x = 31/1.5 = 20.66$ we get $S_{p \to \infty} = 170/6 = 28.33$.

Problem 3 (3.0 points) Given the following data structures that models a graph in which each node can have up to 4 neighbour nodes. Vector `g` holds the information for `N` nodes; for each node the `telem` struct stores the weight of the node `w`, an integer to identify each of the 4 possible neighbours (`north`, `east`, `west` and `south`) and a field that is used to traverse the graph (`visited`) .

```
#define N ... /* large value */

typedef struct {
        int w;
        int north, east, west, south; // -1 value to indicate no neighbour
        char visited;
} telem;
telem g[N];
```

The following code processes all the nodes that are reachable from a given node (0 in the invocation from main):

```
int compute (int label, int weight); // heavy computation, does not access the graph

int traverse_reachable (int label) {
        int ret=0, ret1, ret2, ret3, ret4;
        if (label >= 0) {
            if (!g[label].visited) {
                g[label].visited=1;
                ret1 = traverse_reachable (g[label].north);
                ret2 = traverse_reachable (g[label].east);
                ret3 = traverse_reachable (g[label].west);
                ret4 = traverse_reachable (g[label].south);
                ret = compute(label, g[label].w) + ret1 + ret2 + ret3 + ret4;
            }
        }
        return ret;
}

int main() {
```

```
    ...
    int ret = traverse_reachable (0);
    ...
}
```

We ask you to write an *OpenMP* parallel version of the previous code using a *recursive tree task decomposition* strategy. The implementation should maximize parallelism and minimize the possible synchronization overheads. The implementation must also include a task generation control mechanism based on the recursion level. Use *MAX_ DEPTH* as the value for the maximum recursion level for which tasks must be generated.

Solution:

```
typedef struct {
        int w;
        int north, east, west, south;
        char visited;
        omp_lock_t lock;    // new field to protect the access to the node
} telem;
telem g[N];

int traverse_reachable (int label, int d) { // new argument d to control recursion level
        int ret=0, ret1, ret2, ret3, ret4, tmp;

        if (label >= 0) {
             if (!g[label].visited) {
                omp_set_lock (&g[label].lock);
                if (!g[label].visited) {
                    g[label].visited=1;
                    omp_unset_lock (&g[label].lock);

                    if (!omp_in_final()) {
                        #pragma omp task shared(ret1) final (d>= MAX_DEPTH)
                        ret1 = traverse_reachable (g[label].north, d+1);
                        #pragma omp task shared(ret2) final (d>= MAX_DEPTH)
                        ret2 = traverse_reachable (g[label].east, d+1);
                        #pragma omp task shared(ret3) final (d>= MAX_DEPTH)
                        ret3 = traverse_reachable (g[label].west, d+1);
                        #pragma omp task shared(ret4) final (d>= MAX_DEPTH)
                        ret4 = traverse_reachable (g[label].south, d+1);
                        tmp = compute(label, g[label].w);
                        #pragma omp taskwait     // compute does not need to wait for the graph t
                    } else {
                        ret1 = traverse_reachable (g[label].north, d+1);
                        ret2 = traverse_reachable (g[label].east, d+1);
                        ret3 = traverse_reachable (g[label].west, d+1);
                        ret4 = traverse_reachable (g[label].south, d+1);
                        int tmp = compute(label, g[label].w);
                    }
                    ret = tmp + ret1 + ret2 + ret3 + ret4;
                } else {
                    omp_unset_lock (&g[label].lock);
                }
            }
        }
        return ret;
}

int main() {
    ...
    #pragma omp parallel
```

```
    #pragma omp single
    int ret = traverse_reachable (0, 0);
    ...
}
```

Problem 4 (4.0 points) Given the following sequential code:

```
void saxpy(int n, float a, float * x, float * y) {
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

Assume parameters `x` and `y` point to two completely disjoint vectors, first element of each vector aligned to a memory/cache line boundary. Cache line size is 64 bytes and sizeof(float) is 4 bytes. **We ask you to:**

1. Write a first OpenMP parallel version for the previous sequential function that obeys to an *input block geometric data decomposition* strategy, so that each thread accesses to a single block of consecutive elements. The *block geometric decomposition* should minimise the possible load unbalance that occurs when the size of the vectors `n` is not a multiple value of the number of threads. Assume that `n` is large compared to the number of threads.

   Solution:

   ```
   void saxpy(int n, float a, float * x, float * y) {
     #pragma omp parallel
     {
        int nt;
        int my_id;
        int red;
        int i_start, i_end, n_elems;
        nt    = omp_get_num_threads();
        my_id = omp_get_thread_num();
        n_elems = n/nt;
        red     = n%nt;
        i_start = my_id*n_elems;
        i_start = i_start + (my_id<red)?my_id:red;
        i_end   = i_start + n_elems + (my_id<red);
        for (int i = i_start; i < i_end; ++i)
            y[i] = a*x[i] + y[i];
     }
   }
   ```

2. Write a new parallel version that obeys to an *input/output cyclic geometric data decomposition* strategy. Has the load balance improved with respect to the previous version?

   Solution:

   ```
   void saxpy(int n, float a, float * x, float * y) {
     #pragma omp parallel
     {
       int nt;
       int my_id;
       nt    = omp_get_num_threads();
       my_id = omp_get_thread_num();
       for (int i = my_id; i < n; i+=nt)
         y[i] = a*x[i] + y[i];
     }
   }
   ```

   Both implementantions block and cyclic geometric data decomposition have the same load unbalance: maximum 1 element.

3. Assuming the *input/output cyclic geometric data decomposition strategy* in the previous implementation, and that 1) the parallel system is composed of 2 NUMA nodes, each with a single processor and private cache; 2)

the parallel program is executed with 2 threads (i.e. *thread* i in NUMA node i); 3) the operating system makes use of "first touch" at page level to decide the allocation of memory addresses to NUMA nodes, with one line per memory page; 4) main memory, directories and private caches are empty when the function above starts its execution; and 5) the execution of the iterations assigned to the two threads are interleaved in time, as shown in the two leftmost columns in the table in the answer sheet for the first 4 iterations of the loop. Data coherence across NUMA nodes is provided by a write-invalidate MSU directory-based system. Complete this table with the information of the directory entries where elements of vector `x` and `y` are stored.

4. Write a final parallel version that obeys to an *output block-cyclic geometric data decomposition* strategy, so that the overhead associated with the coherence protocol in a NUMA multiprocessor architecture is minimised, i.e. exploiting the data locality of both input and output data. Is the load balance better/worse than the one achieved in the parallel versions in question 1 and 2?

   Solution:

   We set BS to the number of float elements that fits in a cache/memory line. In this way we avoid extra cache misses accessing both vector `x` and `y` and false sharing accessing `y`.

```
#define BS (64/4)
void saxpy(int n, float a, float * x, float * y) {
  #pragma omp parallel
  {
    int nt;
    int my_id;
    nt    = omp_get_num_threads();
    my_id = omp_get_thread_num();
    for (int ii = my_id*BS; ii < n; ii+=nt*BS)
      for (int i = ii; i < max(n,ii+BS); i++)
         y[i] = a*x[i] + y[i];
  }
}
```

   Block cyclic geometric data decomposition may have `BS` elements of load unbalance. Therefore, it is worse than before.

Student name: .....

Answer sheet for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status)
0	1	1 - read x[1]			
		1 - read y[1]			
		1 - write y[1]			
1	0	0 - read x[0]			
		0 - read y[0]			
		0 - write y[0]			
2	0	2 - read x[2]			
		2 - read y[2]			
		2 - write y[2]			
3	1	3 - read x[3]			
		3 - read y[3]			
		3 - write y[3]			

Solution for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status bits)
0	1	1 - read x[1]	1	10	S
		1 - read y[1]	1	10	S
		1 - write y[1]	1	10	M
1	0	0 - read x[0]	1	11	S
		0 - read y[0]	1	11	S
		0 - write y[0]	1	01	M
2	0	2 - read x[2]	1	11	S
		2 - read y[2]	1	01	M
		2 - write y[2]	1	01	M
3	1	3 - read x[3]	1	11	S
		3 - read y[3]	1	11	S
		3 - write y[3]	1	10	M

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:**

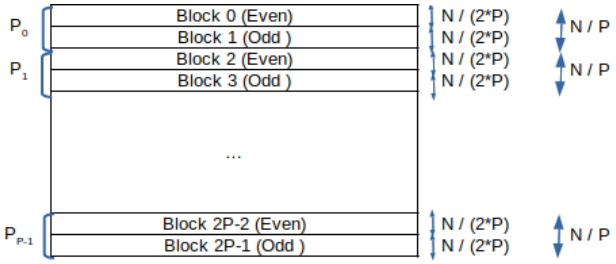
1. Write the expression for the contribution to the total parallel time T_P coming from the parallel computation time T_P^{comp} on an ideal machine with P processors, assuming that: 1) $NB = 2 * P$ perfectly divides the problem size `N`; 2) the execution time for a single iteration of the innermost 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]							

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

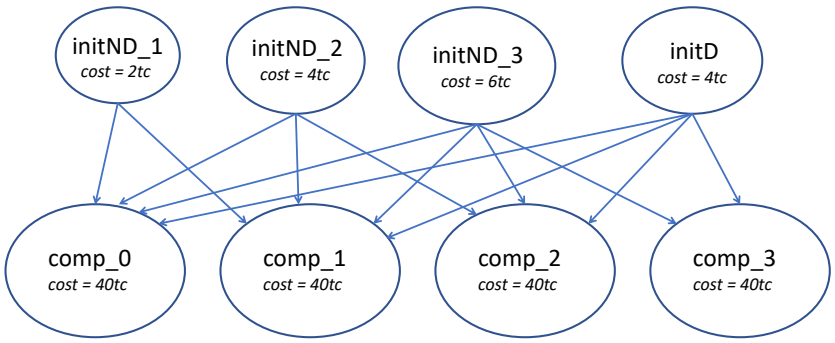
Problem 1 (2 points) Given the following nested loops in a C program instrumented with *Tareador*:

```
#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:

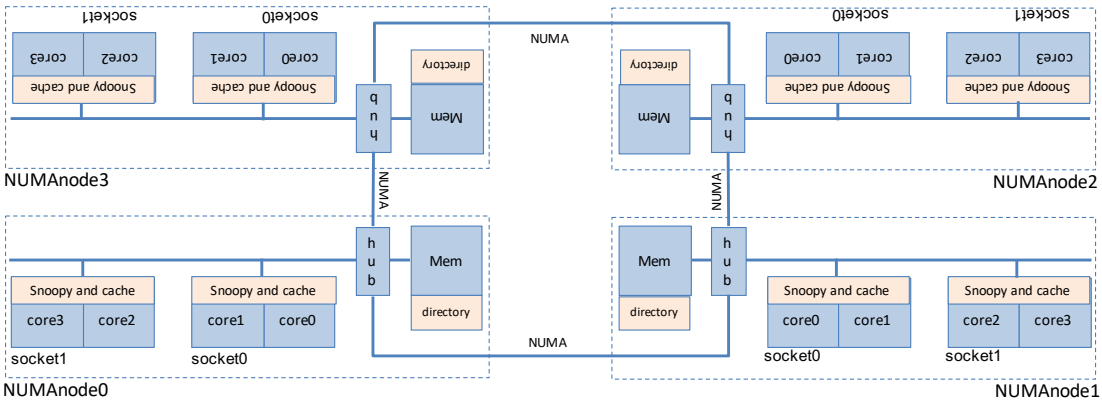
   P_0 = `initND_1, comp_0`

   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.

```
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];

     /* 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 and mergeable clauses 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.

```
typedef struct {
  float max; float min;
} min_max_t;

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

        #pragma omp task shared(min_max1) final(depth>=MAX_DEPTH) mergeable
        min_max1 = find_min_max_rec(v, n2, depth+1);

        #pragma omp task shared(min_max2) final(depth>=MAX_DEPTH) mergeable
        min_max2 = find_min_max_rec(v+n2, n2, depth+1);

        #pragma omp taskwait

        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`.