PAR – 1^{st} ONLINE In-Term Exam – Course 2019/20-Q2

April 22^{nd}, 2020

Version A (3.5 points)

Question 1 (0.25 points) Given the following piece of code instrumented with Tareador:

```
int X[n][n], COEFF[n];
...
for (int i = 1; i < n-1; i++) {
    for (int k = 1; k < n-1; k++) {
        tareador_start_task ("loop");
         int tmp = X[i][k+1] + X[i+1][k] - X[i-1][k] - COEFF[k];
         X[i][k] = tmp/4;
         tareador_end_task ("loop");
    }
}
```

Assume: 1) n is very large; 2) variables i, k and tmp are stored in memory; and 3) the following distribution of matrix X elements to P processors: by rows ($n \div P$ consecutive rows per processor).

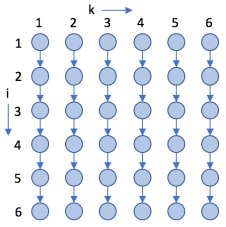
For example, the figure below illustrates the matrix distribution for $P = 4$ and $n = 8$.

P_0	[0,0]	[0,1]	[0,2]	[0,3]	[0,4]	[0,5]	[0,6]	[0,7]
	[1,0]	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]
P_1	[2,0]	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]
	[3,0]	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]
P_2	[4,0]	[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]
	[5,0]	[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]
P_3	[6,0]	[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]
	[7,0]	[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]

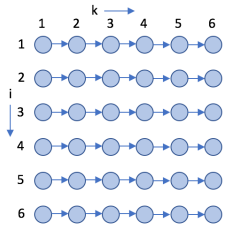
We ask you to:

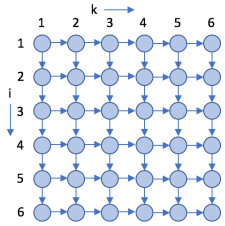
Indicate which of the following **Task Dependence Graph (TDG)** would be the one generated by Tareador.

- TDG (a)



- TDG (b)



- TDG (c) 

- None of the above TDG.

Solution:

The TDG that would be generated by Tareador is (a). In each iteration of the loop, to perform the calculation of the the element X[i][k], the task (i,k) has to perform memory accesses to the elements: X[i][k+1], X[i+1][k], X[i-1][k] and COEFF[k]. After that, the element X[i-1][k] has to be already updated. For this reason task(i,k] can not start until task(i-1,k) finishes.

Question 2 (1.0 point) Assuming each task has a cost of t_c time units, obtain the expression for T_1 and T_∞ as a function of the problem size n. How many processor P_{min} are needed to guarantee that T_∞ can be reached?. **Briefly justify** how did you obtain these expressions.

Solution:

Assuming that n is very large, we can approximate $n-2$ to n. In this way, we can say that the total number of nodes in the TDG is $n \times n$. $T_1 = (n \times n) \times t_c$

The critical path is any column from the first to the last row. As we can observe in the TDG, all the tasks in a row can execute concurrently. $T_\infty = n \times t_c$

As all the tasks in a row can start and progress execution simultaneously, the minimum number of processors needed to reach T_∞ is $P_{min} = n$.

Question 3 (0.25 points) Assume that each processor executes all the tasks that computes the elements of the matrix that are stored in its memory, according to the data distribution indicated in the first question. We ask you to indicate in which order each processor should execute its assigned tasks in order to maximise parallelism (i.e. minimise execution time) while preserving data dependencies. Briefly justify your answer.

Solution:

The tasks processing border rows need to exchange rows with neighbour processors. In particular, the last row assigned to each processor P_i, being $0 < i < n-2$, is accessed from processor P_{i+1}, but it needs the updated values. For this reason, the sooner border elements are updated, the more parallelism is achieved. In consequence, the best order to maximise parallelism preserving data dependencies is traversing the elements by columns at each processor. In consequence, the best order to maximise parallelism preserving data dependencies is traversing the elements by columns at each processor.

Question 4 (1.0 point) Write the expression for the execution time T_p (include only execution time) as a function of the number of processors P and the problem size n. Briefly justify how did you obtain the expression.

Solution:

The calculation of T_p involves the time needed for last processor to execute sequentially all its assigned tasks plus the time it is delayed until it can start execution because of dependencies.

The total execution time per processor P_i is: $t_i = t_c \times (n \times (n \div P))$

Let's name $t_{col} = t_c \times (n \div P)$ time units, the execution time of a column of tasks in a processor. Traversing elements by columns means that the first task in a processor P_i can start execution with a delay $delay = t_{col} \times (P_i - 1)$ time units. In consequence, tasks at processor $P-1$ can start execution after $t_{col} \times (P-2)$ time units.

So finally the expression for the execution time T_p can be written as follows: $T_p = (t_{col} \times (P-2) + t_c \times (n \times (n \div P)) = = ((t_c \times (n \div P) \times (P-2)) + t_c \times (n \times (n \div P))$

Question 5 (1.0 point) Assume a synchronization cost of t_{sync} time units: when a tasks finishes it takes t_{sync} to signal ALL its successor tasks. Obtain the synchronization overhead that would be added to the expression for T_p as a function of the number of processors P, the problem size n and t_{sync}. Briefly justify how did you obtain the expression.

Solution:

We have to take into account the overhead added to the tasks in the critical path. All the tasks in the critical path have a sucessor task except for the last one:

Tasks from the last processor with a sucessor: $((n \times (n \div P)) - 1)$. And the tasks that delay the execution of the last processor, which are composed by $(n \div P) \times (P-2)$. So the final expression for the overhead of synchronization is:

$T_{overhead} = (((n \times (n \div P)) - 1) + (n \div P) \times (P-2)) \times t_{sync}$

Version B (3.5 points)

Question 1 (0.25 points) Given the following piece of code instrumented with Tareador:

```
int X[n][n], COEFF[n];
...
for (int i = 1; i < n-1; i++) {
    for (int k = 1; k < n-1; k++) {
        tareador_start_task ("loop");
         int tmp = X[i][k+1] + X[i+1][k] - X[i][k-1] - COEFF[k];
         X[i][k] = tmp/4;
         tareador_end_task ("loop");
    }
}
```

Assume: 1) n is very large; 2) variables i, k and tmp are stored in memory; and 3) the following distribution of matrix X elements to P processors: by columns ($n \div P$ consecutive columns per processor).

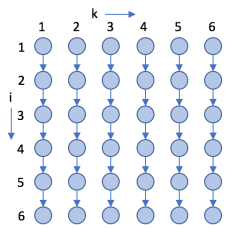
For example, the figure below illustrates the matrix distribution for $P = 4$ and $n = 8$.

[0,0]	[0,1]	[0,2]	[0,3]	[0,4]	[0,5]	[0,6]	[0,7]
[1,0]	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]
[2,0]	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]
[3,0]	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]
[4,0]	[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]
[5,0]	[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]
[6,0]	[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]
[7,0]	[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]
P_0		P_1		P_2		P_3	

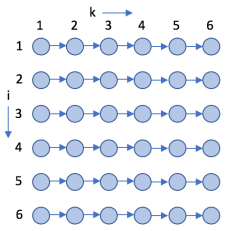
We ask you to:

Indicate which of the following **Task Dependence Graph (TDG)** would be the one generated by Tareador.

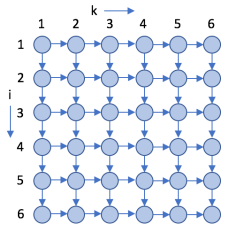
- TDG (a)



- TDG (b)



- TDG (c)



- None of the above TDG.

Solution:

The TDG that would be generated by Tareador is (b). In each iteration of the loop, to perform the calculation of the the element X[i][k], the task (i,k) has to perform memory accesses to the elements: X[i][k+1], X[i+1][k], X[i1][k-1] and COEFF[k]. After that, the element X[i][k-1] has to be already updated. For this reason task(i,k] cannot start until task(i,k-1) finishes.

Question 2 (1.0 point) Assuming each task has a cost of t_c time units, obtain the expression for T_1 and T_∞ as a function of the problem size n. How many processor P_{min} are needed to guarantee that T_∞ can be reached?. **Briefly justify** how did you obtain these expressions.

Solution:

Assuming that n is very large, we can approximate $n-2$ to n. In this way, we can say that the total number of nodes in the TDG is $n \times n$. $T_1 = (n \times n) \times t_c$

The critical path is any column from the first to the last row. As we can observe in the TDG, all the tasks in a column can execute concurrently. $T_\infty = n \times t_c$

As all the tasks in a row can start and progress execution simultaneously, the minimum number of processors needed to reach T_∞ is $P_{min} = n$.

Question 3 (0.25 points) Assume that each processor executes all the tasks that computes the elements of the matrix that are stored in its memory, according to the data distribution indicated in the first question. We ask you to indicate in which order each processor should execute its assigned tasks in order to maximise parallelism (i.e. minimise execution time) while preserving data dependencies. Briefly justify your answer.

Solution:

The tasks processing border columns need to exchange columns with neighbour processors. In particular, the last column assigned to each processor P_i, being $0 < i < n-2$, is accessed from processor P_{i+1}, but it needs the updated values. For this reason, the sooner border elements are updated, the more parallelism is achieved. In consequence, the best order to maximise parallelism preserving data dependencies is traversing the elements by rows at each processor.

Question 4 (1.0 point) Write the expression for the execution time T_p (include only execution time) as a function of the number of processors P and the problem size n. Briefly justify how did you obtain the expression.

Solution:

The calculation of T_p involves the time needed for last processor to execute sequentially all its assigned tasks

plus the time it is delayed until it can start execution because of dependencies.

The total execution time per processor P_i is: $t_i = t_c \times (n \times (n \div P))$

Let's name $t_{row} = t_c \times (n \div P)$ time units, the execution time of a row of tasks in a processor. Traversing elements by rows means that the first task in a processor P_i can start execution with a delay $delay = t_{col} \times (P_i - 1)$ time units. In consequence, tasks at processor $P - 1$ can start execution after $t_{col} \times (P - 2)$ time units.

So finally the expression for the execution time T_p can be written as follows: $T_p = (t_{col} \times (P-2) + t_c \times (n \times (n \div P)) = = ((t_c \times (n \div P) \times (P-2)) + t_c \times (n \times (n \div P))$

Question 5 (1.0 point) Assume a synchronization cost of t_{sync} time units: when a tasks finishes it takes t_{sync} to signal ALL its successor tasks. Obtain the synchronization overhead that would be added to the expression for T_p as a function of the number of processors P, the problem size n and t_{sync}. Briefly justify how did you obtain the expression.

Solution:

We have to take into account the overhead added to the tasks in the critical path. All the tasks in the critical path have a sucessor task except for the last one:

Tasks from the last processor with a sucessor: $((n \times (n \div P)) - 1)$. And the tasks that delay the execution of the last processor, which are composed by $(n \div P) \times (P - 2)$. So the final expression for the overhead of synchronization is:

$T_{overhead} = (((n \times (n \div P)) - 1) + (n \div P) \times (P - 2)) \times t_{sync}$

Version C (3.5 points)

Question 1 (0.25 points) Given the following piece of code instrumented with Tareador:

```
int X[n][n], COEFF[n];
...
for (int i = 1; i < n-1; i++) {
    for (int k = 1; k < n-1; k++) {
        tareador_start_task ("loop");
         int tmp = X[i][k+1] + X[i][k-1] - X[i-1][k] - COEFF[k];
         X[i][k] = tmp/4;
         tareador_end_task ("loop");
    }
}
```

Assume: 1) n is very large; 2) variables i, k and tmp are stored in memory; and 3) the following distribution of matrix X elements to P processors: by rows ($n \div P$ consecutive rows per processor).

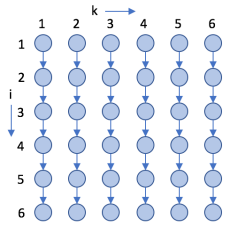
For example, the figure below illustrates the matrix distribution for $P = 4$ and $n = 8$.

P0	[0,0]	[0,1]	[0,2]	[0,3]	[0,4]	[0,5]	[0,6]	[0,7]
	[1,0]	[1,1]	[1,2]	[1,3]	[1,4]	[1,5]	[1,6]	[1,7]
P1	[2,0]	[2,1]	[2,2]	[2,3]	[2,4]	[2,5]	[2,6]	[2,7]
	[3,0]	[3,1]	[3,2]	[3,3]	[3,4]	[3,5]	[3,6]	[3,7]
P2	[4,0]	[4,1]	[4,2]	[4,3]	[4,4]	[4,5]	[4,6]	[4,7]
	[5,0]	[5,1]	[5,2]	[5,3]	[5,4]	[5,5]	[5,6]	[5,7]
P3	[6,0]	[6,1]	[6,2]	[6,3]	[6,4]	[6,5]	[6,6]	[6,7]
	[7,0]	[7,1]	[7,2]	[7,3]	[7,4]	[7,5]	[7,6]	[7,7]

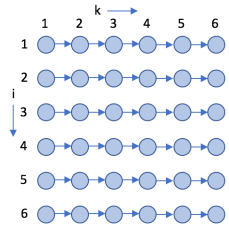
We ask you to:

Indicate which of the following **Task Dependence Graph (TDG)** would be the one generated by Tareador.

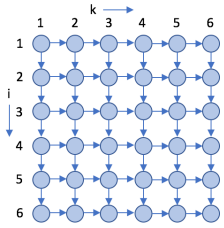
- TDG (a)



- TDG (b)



- TDG (c)



- None of the above TDG.

Solution:

The TDG that would be generated by Tareador is (c). In each iteration of the loop, to perform the calculation of the the element X[i][k], the task (i,k) has to perform memory accesses to the elements: X[i][k+1], X[i][k-1], X[i-1][k] and COEFF[k]. After that, the elements X[i][k-1] and X[i-1][k] have to be already updated. For this reason task(i,k] cannot start until tasks (i,k-1) and (i-1,k) finish.

Question 2 (1.0 point) Assuming each task has a cost of t_c time units, obtain the expression for T_1 and T_∞ as a function of the problem size n. How many processor P_{min} are needed to guarantee that T_∞ can be reached?. **Briefly justify** how did you obtain these expressions.

Solution:

Assuming that n is very large, we can approximate $n-2$ to n. In this way, we can say that the total number of nodes in the TDG is $n \times n$. $T_1 = (n \times n) \times t_c$

The only task that does not need any updated value is task (1,1), and the only task that does not produce any element to be consumed later by a task is task (n-2,n-2). Assuming n-2 can be approximated to n, then we can name the last task (n,n). In summary, the critical path is any that goes from task (1,1) to task (n,n) preserving dependencies. As we can make progress going "down" or to the "right", the length of the critical path is finally $n + n - 1$. $T_\infty = (2.n - 1) \times t_c$

The execution progress is wave-front type, which means tasks belonging to the same reverse diagonal can execute concurrently, then $P_{min} = n$.

Question 3 (0.25 points) Assume that each processor executes all the tasks that computes the elements of the matrix that are stored in its memory, according to the data distribution indicated in the first question. We ask you to indicate in which order each processor should execute its assigned tasks in order to maximise parallelism (i.e. minimise execution time) while preserving data dependencies. Briefly justify your answer.

Solution:

The tasks processing border rows of the bunch of rows assigned to each processor need to exchange rows from adjacent processors. In particular, the last row at each processor P_i, being $0 < i < n-2$, is accessed when updated by processor P_{i+1}. For this reason, the sooner each element of this row is updated, the more parallelism is achieved. In consequence, the best order to maximise parallelism preserving data dependencies is traversing the elements by columns at each processor.

Question 4 (1.0 point) Write the expression for the execution time T_p (include only execution time) as a function of the number of processors P and the problem size n. Briefly justify how did you obtain the expression.

Solution:

The total execution time per processor P_i is: $t_i = t_c \times (n \times (n \div P))$

Traversing elements by columns means that tasks in consecutive processors can start with a delay $d = t_c \times (n \div P)$ time units. In consequence, tasks at processor P-1 can start execution after $d \times (P-1)$ time units.

So finally the expression for the execution time T_p can be written as follows: $T_p = ((t_c \times (n \div P)) \times P - 1) + t_c \times (n \times (n \div P))$

Question 5 (1.0 point) Assume a synchronization cost of t_{sync} time units: when a tasks finishes it takes t_{sync} to signal ALL its successor tasks. Obtain the synchronization overhead that would be added to the expression for T_p as a function of the number of processors P, the problem size n and t_{sync}. Briefly justify how did you obtain the expression.

Solution:

We have to take into account the overhead added to the tasks in the critical path. All the tasks in the critical path have a sucessor task except for the last one which are:

Tasks from the last processor with a sucessor: $((n \times (n \div P)) - 1)$. And the tasks that delay the execution of the last processor, which are composed by $(n \div P) \times (P-2)$. So the final expression for the overhead of synchronization is:

$T_{overhead} = (((n \times (n \div P)) - 1) + (n \div P) \times (P-2)) \times t_{sync}$

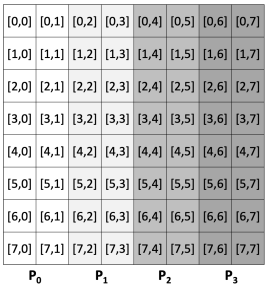
Version D (3.5 points)

Question 1 (0.25 points) Given the following piece of code instrumented with Tareador:

```
int X[n][n], COEFF[n];
...
for (int i = 1; i < n-1; i++) {
    for (int k = 1; k < n-1; k++) {
        tareador_start_task ("loop");
         int tmp = X[i][k+1] + X[i][k-1] - X[i-1][k] - COEFF[k];
         X[i][k] = tmp/4;
         tareador_end_task ("loop");
    }
}
```

Assume: 1) n is very large; 2) variables i, k and tmp are stored in memory; and 3) the following distribution of matrix X elements to P processors: by rows ($n \div P$ consecutive rows per processor).

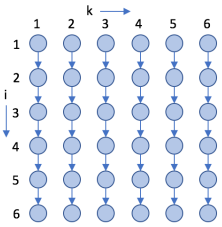
For example, the figure below illustrates the matrix distribution for $P = 4$ and $n = 8$.



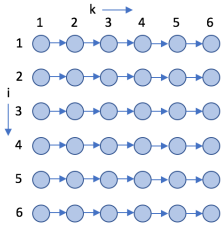
We ask you to:

Indicate which of the following **Task Dependence Graph (TDG)** would be the one generated by Tareador.

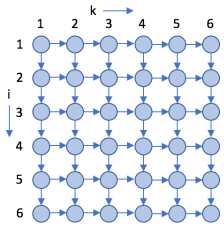
- TDG (a)



- TDG (b)



- TDG (c)



- None of the above TDG.

Solution:

The TDG that would be generated by Tareador is (c). In each iteration of the loop, to perform the calculation of the the element X[i][k], the task (i,k) has to perform memory accesses to the elements: X[i][k+1], X[i][k-1], X[i-1][k] and COEFF[k]. After that, the elements X[i][k-1] and X[i-1][k] have to be already updated. For this reason task(i,k] cannot start until tasks (i,k-1) and (i-1,k) finish.

Question 2 (1.0 point) Assuming each task has a cost of t_c time units, obtain the expression for T_1 and T_∞ as a function of the problem size n. How many processor P_{min} are needed to guarantee that T_∞ can be reached?. **Briefly justify** how did you obtain these expressions.

Solution:

Assuming that n is very large, we can approximate $n-2$ to n. In this way, we can say that the total number of nodes in the TDG is $n \times n$. $T_1 = (n \times n) \times t_c$

The only task that does not need any updated value is task (1,1), and the only task that does not produce any element to be consumed later by a task is task (n-2,n-2). Assuming n-2 can be approximated to n, then

we can name the last task (n,n). In summary, the critical path is any that goes from task (1,1) to task (n,n) preserving dependencies. As we can make progress going "down" or to the "right", the length of the critical path is finally $n + n - 1$. $T_\infty = (2.n - 1) \times t_c$

The execution progress is wave-front type, which means tasks belonging to the same reverse diagonal can execute concurrently, then $P_{min} = n$.

Question 3 (0.25 points) Assume that each processor executes all the tasks that computes the elements of the matrix that are stored in its memory, according to the data distribution indicated in the first question. We ask you to indicate in which order each processor should execute its assigned tasks in order to maximise parallelism (i.e. minimise execution time) while preserving data dependencies. Briefly justify your answer.

Solution:

The tasks processing border rows of the bunch of rows assigned to each processor need to exchange rows from adjacent processors. In particular, the last column at each processor P_i, being $0 < i < n-2$, is accessed when updated by processor P_{i+1}. For this reason, the sooner each element of this column is updated, the more parallelism is achieved. In consequence, the best order to maximise parallelism preserving data dependencies is traversing the elements by rows at each processor.

Question 4 (1.0 point) Write the expression for the execution time T_p (include only execution time) as a function of the number of processors P and the problem size n. Briefly justify how did you obtain the expression.

Solution:

The total execution time per processor P_i is: $t_i = t_c \times (n \times (n \div P))$

Traversing elements by rows means that tasks in consecutive processors can start with a delay $d = t_c \times (n \div P)$ time units. In consequence, tasks at processor P-1 can start execution after $d \times (P - 1)$ time units.

So finally the expression for the execution time T_p can be written as follows: $T_p = ((t_c \times (n \div P)) \times P - 1) + t_c \times (n \times (n \div P))$

Question 5 (1.0 point) Assume a synchronization cost of t_{sync} time units: when a tasks finishes it takes t_{sync} to signal ALL its successor tasks. Obtain the synchronization overhead that would be added to the expression for T_p as a function of the number of processors P, the problem size n and t_{sync}. Briefly justify how did you obtain the expression.

Solution:

We have to take into account the overhead added to the tasks in the critical path. All the tasks in the critical path have a sucessor task except for the last one which are:

Tasks from the last processor with a sucessor: $((n \times (n \div P)) - 1)$. And the tasks that delay the execution of the last processor, which are composed by $(n \div P) \times (P - 2)$. So the final expression for the overhead of synchronization is:

$T_{overhead} = (((n \times (n \div P)) - 1) + (n \div P) \times (P - 2)) \times t_{sync}$

Problem 2 - Questions 6 and 7 (2 points)

Dependence analysis:

All tasks A_i are totally independent: generated either with task or taskloop (implicit taskgroup at the end)

The following dependences needed to be enforced in the code:

1. Between each task A_i and the corresponding task B_i due to the sharing of l[i]
2. Among tasks B_i due to the sharing of variable x
3. Between tasks B_i and task C due to sharing of variable x

Program needs to wait for termination of task C

The following mechanisms in OpenMP are used to enforce dependences in the different code versions:

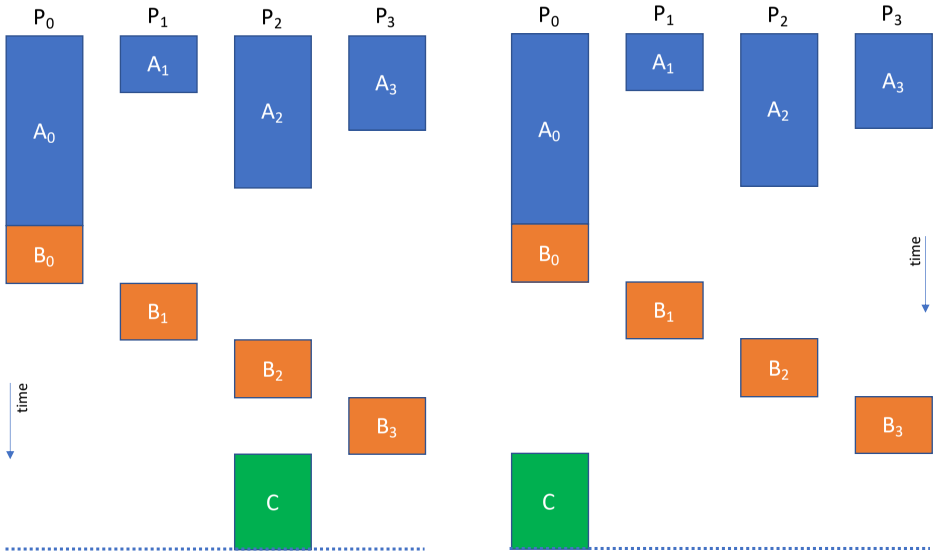
	A_i to B_i	B_i to B_{i+1}	B_i to C	C to end
Code 1	depend on l[i]	depend on x	depend on x	taskwait
Code 2	depend on l[i]	atomic on x	taskwait	taskwait
Code 3	taskwait	atomic on x	taskwait	taskwait
Code 4	implicit taskgroup	critical		taskgroup
Code 5	depend on l[i]	reduction on x	taskgroup	taskwait
Code 6	depend on l[i]	reduction on x		taskgroup
Code 7	depend on l[i]	critical	taskwait	taskwait
Code 8	implicit taskgroup	critical	taskgroup	taskwait

1

```
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])                  // Ai
    l[i] = foo1(i);
}

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

#pragma omp task shared(x) depend(inout:x)              // C
x += foo3(l);
#pragma omp taskwait
…
```

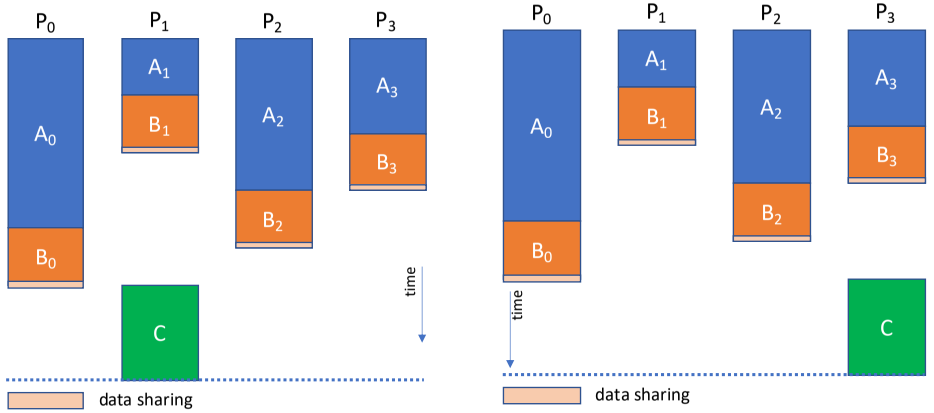


2

```
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i]).                 // Ai
    l[i] = foo1(i);
}

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

#pragma omp task shared(x)                              // C
x += foo3(l);
#pragma omp taskwait
…
```

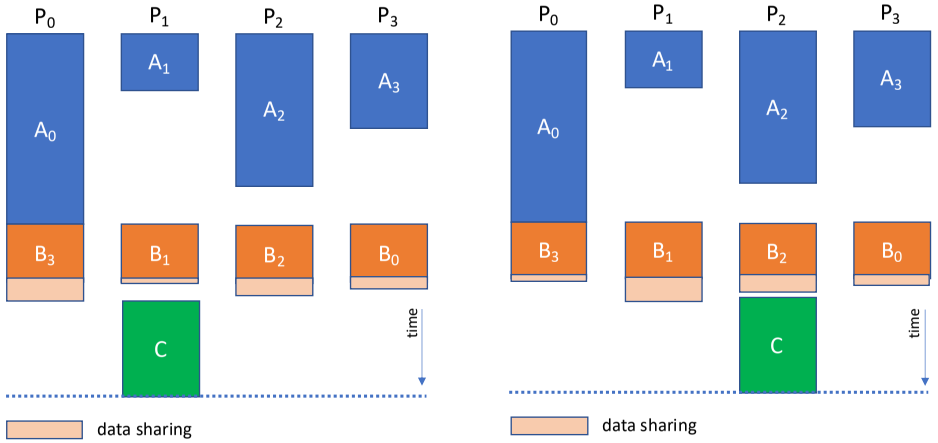


3

```
for (int i=0; i<4; i++) {
    #pragma omp task                          // Ai
    l[i] = foo1(i);
}
#pragma omp taskwait

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

#pragma omp task shared(x)                    // C
x += foo3(l);
#pragma omp taskwait
…
```

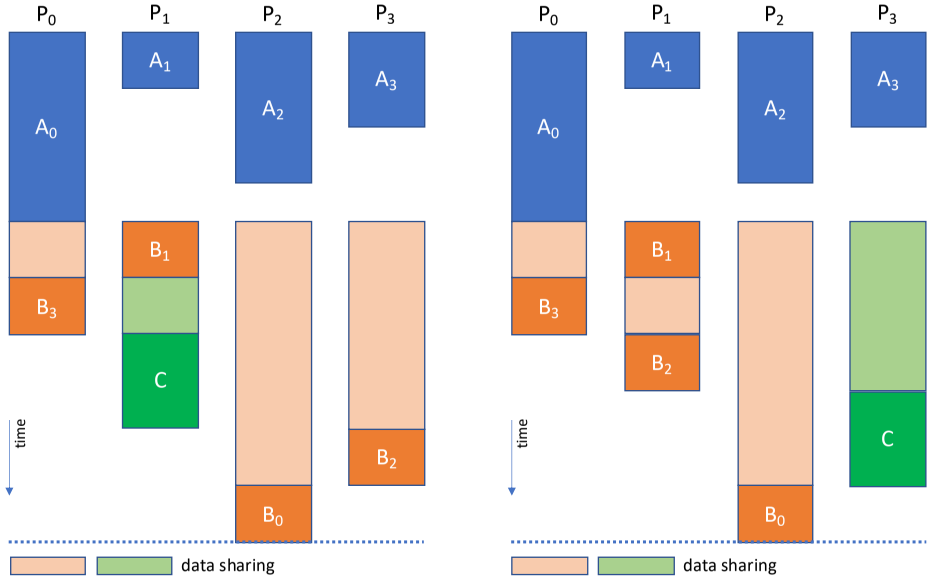


4

```
#pragma omp taskloop grainsize(1)             // Ai
for (int i=0; i<4; i++) {
    l[i] = foo1(i);
}

#pragma omp taskgroup
{
    for (int i=0; i<4; i++) {
        #pragma omp task shared(x)            // Bi
        #pragma omp critical
        x += foo2(i, l[i]);
    }

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

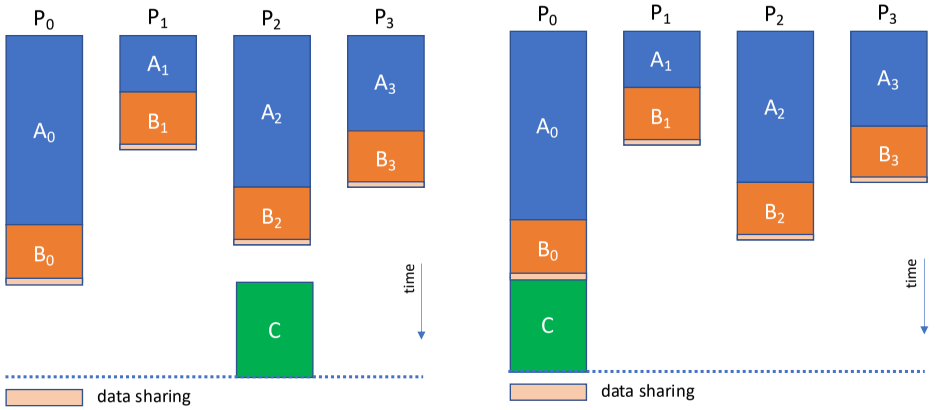


5

```
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])                  // Ai
    l[i] = foo1(i);
}

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

#pragma omp task shared(x)                              // C
x += foo3(l);
#pragma omp taskwait
…
```

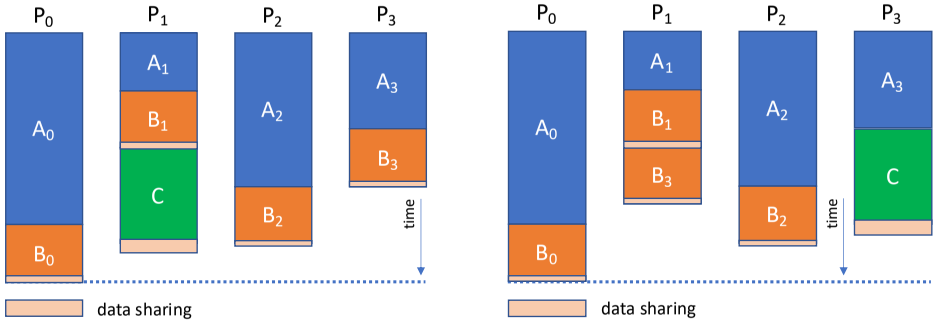


6

```
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])                  // Ai
    l[i] = foo1(i);
}

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

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

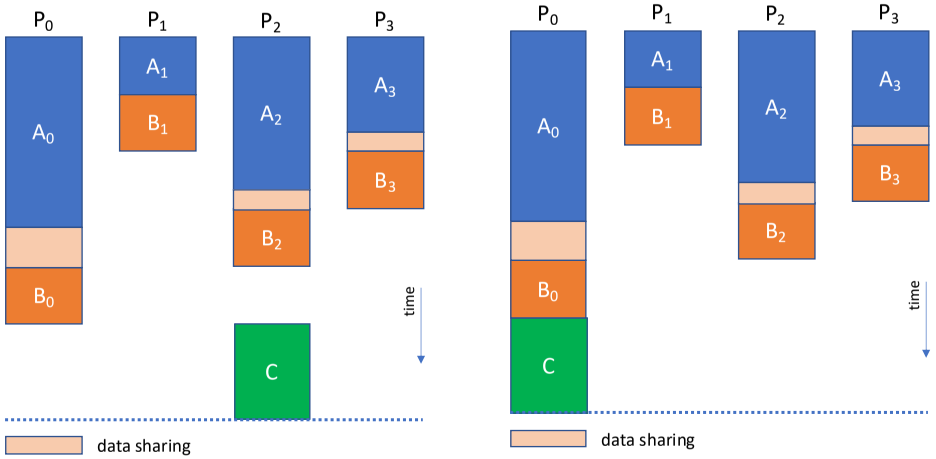


7

```
for (int i=0; i<4; i++) {
    #pragma omp task depend(out: l[i])                  // Ai
    l[i] = foo1(i);
}

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

#pragma omp task shared(x)                              // C
x += foo3(l);
#pragma omp taskwait
…
```

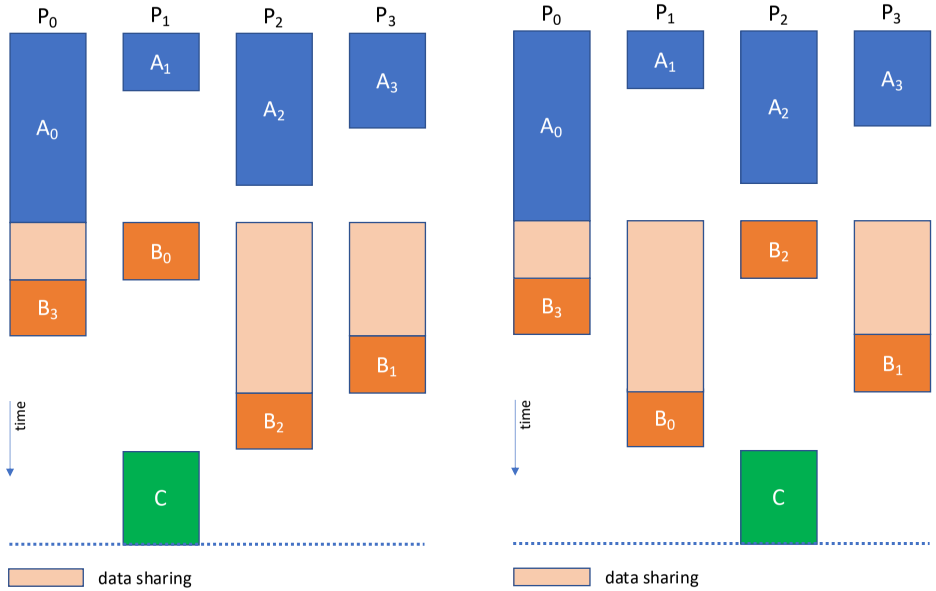


8

```
#pragma omp taskloop grainsize(1)                       // Ai
for (int i=0; i<4; i++) {
    l[i] = foo1(i);
}

#pragma omp taskgroup
for (int i=0; i<4; i++) {
    #pragma omp task shared(x)                          // Bi
    #pragma omp critical
    x += foo2(i, l[i]);
}

#pragma omp task shared(x)                              // C
x += foo3(l);
#pragma omp taskwait
…
```



Problem 3: recursive task decomposition (questions 8–10) (2.5 points)

In this recursive task decomposition problem there are 12 different versions (with some variants): a) Tree and Leaf recursive task decomposition; b) 3 different versions of the recursive sequential code; and c) two different versions for the cut-off mechanism. Regarding the three different recursive sequential codes we have:

- In the "local version", the recursive sequential code returns a value, which is used after the recursive call or base case call.
- In the "global version" the recursive sequential code directly updates a global variable. In this case, there are two possible codes:
  - The "scalar" version, which only updates a variable of a basic C type and scalar.
  - The "vector" version, which updates positions of a globally declared vector.

For brevity, the solutions shown in this document are only for one of the cut-off mechanisms; the other one could be easily extrapolated. The following table shows the section where you can find the solution that you were asked in your specific exam.

Recursive Task Decomposition Versions			
Parallel Strategy	Local	Scalar Global	Vector Global
Tree	Section 1.1	Section 1.2	Section 1.3
Leaf	Section 2.1	Section 2.2	Section 2.3

Table 1: Recursive Task Decomposition versions of the exam

1 Tree Recursive Task Decomposition Strategy

1.1 Version Local variables

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```
int it_PROCESS(int* B, int* J, int* H, unsigned int n) {
   unsigned int i;
   int x_local=0;

   for(i=0; i<n; i++) {
      x_local = x_local + (B[i] - J[i] / H[i]);
   }
   return x_local;
}

int PROCESS(int* B, int* J, int* H, unsigned int n) {
   int x_local=0;

   if (n<(1<<6)) {
      x_local= it_PROCESS(B, J, H, n);
   } else {
      unsigned int i, nNUM_CALLS=n/NUM_CALLS;

      for(i=0; i<NUM_CALLS-1; i++) {
         x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS);
      }
      x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i);
   }
   return x_local;
}

void main() {
   int x_local;
   ...
   x_local = PROCESS(B, J, H, (1<<30));
   ...
}
```

Assuming you have to implement a TREE recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.

- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we have to wait for any task on the recursive calls and their return value

  Data sharing: we have to make local variables be shared to see the return value. In addition, we have to control or avoid any data race condition updating the local value made shared.

- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

  Solution Comments:

  Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors)..

Code Solution for Questions 8 and 10:

The following code show a solution that includes solutions to question 8 and 9 of the quiz. For this solution we will use `omp_in_final` API call and final clause. Following solutions we will use mergeable clause if needed.

```
1
2  int it_PROCESS(int* B, int* J, int* H, unsigned int n) {
3     unsigned int i;
4     int x_local=0;
5
6     for(i=0; i<n; i++) {
7        x_local = x_local + (B[i] - J[i] / H[i]);
8     }
9     return x_local;
10 }
11
12 int PROCESS(int* B, int* J, int* H, unsigned int n, unsigned int DEPTH) {
13    // local vector to avoid data sharing against the same x_local variable
14    int x_local_vect[NUM_CALLS];
15    int x_local=0;
16
17    if (n<(1<<6)) {
18       x_local= it_PROCESS(B, J, H, n);
19    } else {
20       unsigned int i, nNUM_CALLS=n/NUM_CALLS;
21
22       if (!omp_in_final())
23       {
24          for(i=0; i<NUM_CALLS-1; i++) {
25             #pragma omp task shared(x_local_vect) final(DEPTH>=CUTOFF_LEVEL)
26             x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH+1);
27          }
28          #pragma omp task shared(x_local_vect) final(DEPTH>=CUTOFF_LEVEL)
29          x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH+1);
30
31          #pragma omp taskwait
32          for(i=0; i<NUM_CALLS; i++)
33             x_local += x_local_vect[i];
34       }
35       else {
36          for(i=0; i<NUM_CALLS-1; i++) {
37             x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH);
38          }
39          x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH);
40       }
41    }
42    return x_local;
43 }
44
45 void main() {
46    int x_local;
47    ...
48    #pragma omp parallel
49    #pragma omp single
50    x_local = PROCESS(B, J, H, (1<<30), 0);
51    ...
52 }
```

1.2 Version Scalar Global Variable

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```
unsigned int u_global=1;

void it_JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   unsigned int i;

   for(i=0; i<n; i++) {
      u_global = u_global * (C[i] - D[i] * G[i]);
   }
}

void JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   if (n>(1<<10)) {
      unsigned int i, n4=n/4;

      JoEmQuedoACasa(C+n4*0, D+n4*0, G+n4*0, n4);
      JoEmQuedoACasa(C+n4*1, D+n4*1, G+n4*1, n4);
      JoEmQuedoACasa(C+n4*2, D+n4*2, G+n4*2, n4);
      JoEmQuedoACasa(C+n4*3, D+n4*3, G+n4*3, n-n4*3);
   } else {
      it_JoEmQuedoACasa(C, D, G, n);
   }
}
```

Assuming you have to implement a TREE recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.
- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we do not have to wait for any task on the recursive calls since all updates are done in the global variable

  Data sharing: We have to avoid data race conditions, controling any access to global variable. In order to reduce the number of synchronizations we have created a local variable to finally use it to update global variable.
- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

  Solution Comments:

  Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors).

Code Solution for Questions 8 and 10:

The following code show a solution that includes solutions to question 8 and 9 of the quiz. For this solution we will use mergeable clause. For the first vesion of this problem we use `omp_in_final` API of OpenMP.

```
unsigned int u_global=1;

void it_JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   unsigned int i;

   unsigned int u_local=1;
   for(i=0; i<n; i++) {
      u_local= u_local* (C[i] - D[i] * G[i]);
   }
   #pragma omp atomic
   u_global= u_global* u_local;
}

// Solution with mergeable (considering it is properly implemented)

```

```
void JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n, unsigned int DEPTH) {
   if (n>(1<<10)) {
      unsigned int i, n4=n/4;
      #pragma omp task final(DEPTH>CUTOFF_LEVEL) mergeable
      JoEmQuedoACasa(C+n4*0, D+n4*0, G+n4*0, n4, DEPTH+1);
      #pragma omp task final(DEPTH>CUTOFF_LEVEL) mergeable
      JoEmQuedoACasa(C+n4*1, D+n4*1, G+n4*1, n4, DEPTH+1);
      #pragma omp task final(DEPTH>CUTOFF_LEVEL) mergeable
      JoEmQuedoACasa(C+n4*2, D+n4*2, G+n4*2, n4, DEPTH+1);
      #pragma omp task final(DEPTH>CUTOFF_LEVEL) mergeable
      JoEmQuedoACasa(C+n4*3, D+n4*3, G+n4*3, n-n4*3, DEPTH+1);
   } else {
      it_JoEmQuedoACasa(C, D, G, n);
   }
}

void main() {
   ...
   #pragma omp parallel
   #pragma omp single
   JoEmQuedoACasa(C, D, G, (1<<21), 0);
   ...
}
```

1.3 Version Global Vector Variable

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```

double x_global[MAX_MAP];

void it_JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
   unsigned int i;

   for(i=0; i<n; i++) {
      unsigned int temp_result= map_update(G[i]*H[i]-D[i]);
      PROCESS(&x_global[temp_result]);
   }
}

void JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
   if (n<=(1<<10)) {
      it_JoEmQuedoACasa(G, H, D, n);
   } else {
      unsigned int i, n4=n/4;

      JoEmQuedoACasa(G+n4*0, H+n4*0, D+n4*0, n4);
      JoEmQuedoACasa(G+n4*1, H+n4*1, D+n4*1, n4);
      JoEmQuedoACasa(G+n4*2, H+n4*2, D+n4*2, n4);
      JoEmQuedoACasa(G+n4*3, H+n4*3, D+n4*3, n-n4*3);
   }
}

void main() {
   ...
   JoEmQuedoACasa(G, H, D, (1<<18));
   ...
}
```

Assuming you have to implement a TREE recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.
- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we do not have to wait for any task on the recursive calls since all updates are done in the global variable

  Data sharing: We have to avoid data race conditions, controling any access to global variable. In order to reduce the number of synchronizations and allow some parallelism it is better to use locks based on the position of the global vector to be updated.
- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

Solution Comments:

Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors).

Code Solution for Questions 8 and 10:

The following code show a solution that includes solutions to question 8 and 9 of the quiz. For this solution we will use mergeable clause. For the first vesion of this problem we use `omp_in_final` API of OpenMP.

```
1
2  double x_global[MAX_MAP];
3  omp_lock_t lock_global[MAX_MAP];
4
5  void it_JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
6     unsigned int i;
7
8     for(i=0; i<n; i++) {
9        unsigned int temp_result= map_update(G[i]*H[i]-D[i]);
10       omp_set_lock(&lock_global[temp_result]);
11       PROCESS(&x_global[temp_result]);
12       omp_unset_lock(&lock_global[temp_result]);
13    }
14 }
15
16
17 // Solution with mergeable (considering it is properly implemented)
18 void JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n, DEPTH) {
19    if (n<=(1<<10)) {
20       it_JoEmQuedoACasa(G, H, D, n);
21    } else {
22       unsigned int i, n4=n/4;
23       #pragma omp task final(DEPTH>=CUTOFF_LEVEL) mergeable
24       JoEmQuedoACasa(G+n4*0, H+n4*0, D+n4*0, n4, DEPTH+1);
25       #pragma omp task final(DEPTH>=CUTOFF_LEVEL) mergeable
26       JoEmQuedoACasa(G+n4*1, H+n4*1, D+n4*1, n4, DEPTH+1);
27       #pragma omp task final(DEPTH>=CUTOFF_LEVEL) mergeable
28       JoEmQuedoACasa(G+n4*2, H+n4*2, D+n4*2, n4, DEPTH+1);
29       #pragma omp task final(DEPTH>=CUTOFF_LEVEL) mergeable
30       JoEmQuedoACasa(G+n4*3, H+n4*3, D+n4*3, n-n4*3, DEPTH+1);
31    }
32 }
33
34 void main() {
35    ...
36
37    int i;
38
39    for (i=0; i<MAX_MAP; i++)
40       omp_init_lock(&lock_global[i]);
41
42    #pragma omp parallel
43    #pragma omp single
44    JoEmQuedoACasa(G, H, D, (1<<18), 0);
45    ...
46
47    for (i=0; i<MAX_MAP; i++)
48       omp_destroy_lock(&lock_global[i]);
49 }
```

2 Leaf Recursive Task Decomposition Strategy

2.1 Version Local variables

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```
1  int it_PROCESS(int* B, int* J, int* H, unsigned int n) {
2     unsigned int i;
3     int x_local=0;
4
5     for(i=0; i<n; i++) {
6        x_local = x_local + (B[i] - J[i] / H[i]);
7     }
8     return x_local;
9  }
10
11 int PROCESS(int* B, int* J, int* H, unsigned int n) {
12    int x_local=0;
13
14    if (n<(1<<6)) {
15       x_local= it_PROCESS(B, J, H, n);
16    } else {
17       unsigned int i, nNUM_CALLS=n/NUM_CALLS;
18
19       for(i=0; i<NUM_CALLS-1; i++) {
20          x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS);
21       }
22       x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i);
23    }
```

```
   return x_local;
}

void main() {
   int x_local;
   ...
   x_local = PROCESS(B, J, H, (1<<30));
   ...
}
```

Assuming you have to implement a LEAF recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.
- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we have to wait for any task on the recursive calls and their return value

  Data sharing: we have to make local variables be shared to see the return value. In addition, we have to control or avoid any data race condition updating the local value made shared.
- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

  Solution Comments:

  Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors).

  Cutoff control at leaf strategy should also control the cutoff level at the recursive part of the code.

The following codes show two different solutions for questions 8 and 9 of the quiz.

2.1.1 Solution with `taskloop` in the iterative function

This code does exploit thread level parallelism if the tasks are created at the leaf level.

```
// We have added a new parameter, DEPTH, to control if it is necessary to create or not tasks.
int it_PROCESS(int* B, int* J, int* H, unsigned int n, unsigned int DEPTH) {
   unsigned int i;
   int x_local=0;

   if (DEPTH<CUTOFF_LEVEL) {
      // Task loop in order to have several tasks running in parallel and
      // usage of reduction in order to avoid synchronizations within the loop
      // Num tasks clause can have a different value, it is just an example.
      #pragma omp taskloop num_tasks(4) reduction(+:x_local)
      for(i=0; i<n; i++) {
         x_local = x_local + (B[i] - J[i] / H[i]);
      }
   } else
      for(i=0; i<n; i++) {
         x_local = x_local + (B[i] - J[i] / H[i]);
      }

   return x_local;
}

int PROCESS(int* B, int* J, int* H, unsigned int n, unsigned int DEPTH) {
   int x_local_vect[NUM_CALLS]; // declare local vector to avoid data race condition
   int x_local=0;

   if (n<(1<<6)) {
      x_local= it_PROCESS(B, J, H, n, DEPTH);
   } else {
      unsigned int i, nNUM_CALLS=n/NUM_CALLS;

      // It is important to control which level of recursivity we are to create or not tasks
      if (DEPTH==CUTOFF_LEVEL)  {
         for(i=0; i<NUM_CALLS-1; i++) {
```

```
            #pragma omp task shared(x_local_vect) firsprivate(i)
            x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH+1);
         }
         #pragma omp task shared(x_local_vect) firsprivate(i)
         x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH+1);

         #pragma omp taskwait
         for(i=0; i<NUM_CALLS; i++)
            x_local += x_local_vect[i];
      } else {
         for(i=0; i<NUM_CALLS-1; i++) {
            x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH);
         }
         x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH);
      }
   }
   return x_local;
}

void main() {
   int x_local;
   ...
   #pragma omp parallel
   #pragma omp single
   x_local = PROCESS(B, J, H, (1<<30), 0);
   ...
}
```

2.1.2 Solution without `taskloop` in the iterative function

This code does not exploit thread level parallelism if the tasks are created at the leaf level.

```
int it_PROCESS(int* B, int* J, int* H, unsigned int n) {
   unsigned int i;
   int x_local=0;

   for(i=0; i<n; i++) {
     x_local = x_local + (B[i] - J[i] / H[i]);
   }
   return x_local;
}

int PROCESS(int* B, int* J, int* H, unsigned int n, unsigned int DEPTH) {
   int x_local_vect[NUM_CALLS]; // declare local vector to avoid data race condition
   int x_local=0;

   if (n<(1<<6)) {
     if (DEPTH<CUTOFF_LEVEL) {
         #pragma omp task shared(x_local)
         x_local= it_PROCESS(B, J, H, n, DEPTH);
         // it is not necessary atomic since the store is atomic
         // AND ONLY THIS TASK IS RUNNING AT THE SAME TIME SHARING
         // X_LOCAL :/

         #pragma omp taskwait
     } else
         x_local= it_PROCESS(B, J, H, n, DEPTH);
     } else {
      unsigned int i, nNUM_CALLS=n/NUM_CALLS;
      if (DEPTH==CUTOFF_LEVEL) {
         for(i=0; i<NUM_CALLS-1; i++) {
            #pragma omp task shared(x_local_vect) firsprivate(i)
            x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH+1);
         }
         #pragma omp task shared(x_local_vect) firsprivate(i)
         x_local_vect[i] = PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH+1);

         #pragma omp taskwait
         for(i=0; i<NUM_CALLS; i++)
            x_local += x_local_vect[i];
      } else {
         for(i=0; i<NUM_CALLS-1; i++) {
            x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, nNUM_CALLS, DEPTH);
         }
         x_local = x_local + PROCESS(B+nNUM_CALLS*i, J+nNUM_CALLS*i, H+nNUM_CALLS*i, n-nNUM_CALLS*i, DEPTH);
      }
   }
   return x_local;
}

void main() {
   int x_local;
   ...
   #pragma omp parallel
   #pragma omp single
   x_local = PROCESS(B, J, H, (1<<30), 0);
   ...
}
```

2.2 Version Global variable

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```
unsigned int u_global=1;
```

```
void it_JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   unsigned int i;

   for(i=0; i<n; i++) {
      u_global = u_global * (C[i] - D[i] * G[i]);
   }
}

void JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   if (n>(1<<10)) {
      unsigned int i, n4=n/4;

      JoEmQuedoACasa(C+n4*0, D+n4*0, G+n4*0, n4);
      JoEmQuedoACasa(C+n4*1, D+n4*1, G+n4*1, n4);
      JoEmQuedoACasa(C+n4*2, D+n4*2, G+n4*2, n4);
      JoEmQuedoACasa(C+n4*3, D+n4*3, G+n4*3, n-n4*3);
   } else {
      it_JoEmQuedoACasa(C, D, G, n);
   }
}
```

Assuming you have to implement a LEAF recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.
- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we do not have to wait for any task on the recursive calls since all updates are done in the global variable

  Data sharing: We have to avoid data race conditions, controling any access to global variable. In order to reduce the number of synchronizations we have created a local variable to finally use it to update global variable.
- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

  Solution Comments:

  Cutoff control at leaf strategy should also control the cutoff level at the recursive part of the code.

  Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors).

Code Solution for Questions 8 and 10:

The following code show a solution that includes solutions to question 8 and 9 of the quiz. For this solution we will use mergeable clause. For the first vesion of this problem we use `omp_in_final` API of OpenMP.

```
unsigned int u_global=1;

void it_JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n) {
   unsigned int i;

   unsigned int u_local=1;
   for(i=0; i<n; i++) {
      u_local= u_local* (C[i] - D[i] * G[i]);
   }
   #pragma omp atomic
   u_global= u_global* u_local;
}

// Solution with mergeable (considering it is properly implemented)

void JoEmQuedoACasa(unsigned int* C, unsigned int* D, unsigned int* G, unsigned int n, unsigned int DEPTH) {
   if (n>(1<<10)) {
      unsigned int i, n4=n/4;
      if (DEPTH==CUTOFF_LEVEL)
      {
         #pragma omp task
```

```
            JoEmQuedoACasa(C+n4*0, D+n4*0, G+n4*0, n4, DEPTH+1);
            #pragma omp task
            JoEmQuedoACasa(C+n4*1, D+n4*1, G+n4*1, n4, DEPTH+1);
            #pragma omp task
            JoEmQuedoACasa(C+n4*2, D+n4*2, G+n4*2, n4, DEPTH+1);
            #pragma omp task
            JoEmQuedoACasa(C+n4*3, D+n4*3, G+n4*3, n-n4*3, DEPTH+1);
        }
        else {
            JoEmQuedoACasa(C+n4*0, D+n4*0, G+n4*0, n4, DEPTH+1);
            JoEmQuedoACasa(C+n4*1, D+n4*1, G+n4*1, n4, DEPTH+1);
            JoEmQuedoACasa(C+n4*2, D+n4*2, G+n4*2, n4, DEPTH+1);
            JoEmQuedoACasa(C+n4*3, D+n4*3, G+n4*3, n-n4*3, DEPTH+1);
        }
    } else {
        if (DEPTH <= CUTOFF_LEVEL)
        {
            #pragma omp task
            it_JoEmQuedoACasa(C, D, G, n);
        }
        else it_JoEmQuedoACasa(C, D, G, n);
    }
}

void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    JoEmQuedoACasa(C, D, G, (1<<21), 0);
    ...
}
```

2.3 Version Vector Global variable

Consider the following sequential recursive code, invoked from main program with n=1«30 (Note: (1«m) means 1 shifted m bits left, in other words 2 to the power m):

```

double x_global[MAX_MAP];

void it_JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
    unsigned int i;

    for(i=0; i<n; i++) {
        unsigned int temp_result= map_update(G[i]*H[i]-D[i]);
        PROCESS(&x_global[temp_result]);
    }
}

void JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
    if (n<=(1<<10)) {
        it_JoEmQuedoACasa(G, H, D, n);
    } else {
        unsigned int i, n4=n/4;

        JoEmQuedoACasa(G+n4*0, H+n4*0, D+n4*0, n4);
        JoEmQuedoACasa(G+n4*1, H+n4*1, D+n4*1, n4);
        JoEmQuedoACasa(G+n4*2, H+n4*2, D+n4*2, n4);
        JoEmQuedoACasa(G+n4*3, H+n4*3, D+n4*3, n-n4*3);
    }
}

void main() {
    ...
    JoEmQuedoACasa(G, H, D, (1<<18));
    ...
}
```

Assuming you have to implement a LEAF recursive task decomposition strategy, we ask you to answer the following questions in the open textboxes below.

- Question 8: Write an OpenMP parallel implementation with no cut-off mechanism, i.e. without controlling the number of tasks that are generated or their granularity.
- Question 9: When tasks are created at a recursion level, is there any data sharing/task ordering potential problem that needs to be considered? When parallel tasks execute the function that is called when recursion reaches the base case, is there any data sharing problem caused by the access to the scalar variable within the loop? Reason your answers, explaining either why there are no problems or, in case problems exist, how have you addressed them in your parallel code above.

  Solution Comments:

  Task ordering: we do not have to wait for any task on the recursive calls since all updates are done in the global variable

Data sharing: We have to avoid data race conditions, controling any access to global variable. In order to reduce the number of synchronizations and allow some parallelism it is better to use locks based on the position of the global vector to be updated.

- Question 10: Modify your previous OpenMP parellel implementation in order to reduce parallelization overheads, controlling the generation of tasks based on RECURSION LEVEL (constant CUTOFF_LEVEL).

  Solution Comments:

  Cutoff control at leaf strategy should also control the cutoff level at the recursive part of the code.

  Cutoff control done in the following code is using level of recursivity. In order to reduce the amount of pages of the solution we have decided to show just one. Using size of the vector means that you have to compare with n (number of elements to be processed along the vectors).

Code Solution for Questions 8 and 10:

The following code show a solution that includes solutions to question 8 and 9 of the quiz. For this solution we will use mergeable clause. For the first vesion of this problem we use `omp_in_final` API of OpenMP.

```
1
2  double x_global[MAX_MAP];
3  omp_lock_t lock_global[MAX_MAP];
4
5  void it_JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n) {
6     unsigned int i;
7
8     for(i=0; i<n; i++) {
9        unsigned int temp_result= map_update(G[i]*H[i]-D[i]);
10       omp_set_lock(&lock_global[temp_result]);
11       PROCESS(&x_global[temp_result]);
12       omp_unset_lock(&lock_global[temp_result]);
13    }
14 }
15
16
17 void JoEmQuedoACasa(double* G, double* H, double* D, unsigned int n, DEPTH) {
18    if (n<=(1<<10)) {
19       if (DEPTH<=CUTOFF_LEVEL)
20       {
21          #pragma omp task
22          it_JoEmQuedoACasa(G, H, D, n);
23       }
24       else
25          it_JoEmQuedoACasa(G, H, D, n);
26    } else {
27       unsigned int i, n4=n/4;
28       if (DEPTH==CUTOFF_LEVEL)
29       {
30          #pragma omp task
31          JoEmQuedoACasa(G+n4*0, H+n4*0, D+n4*0, n4, DEPTH+1);
32          #pragma omp task
33          JoEmQuedoACasa(G+n4*1, H+n4*1, D+n4*1, n4, DEPTH+1);
34          #pragma omp task
35          JoEmQuedoACasa(G+n4*2, H+n4*2, D+n4*2, n4, DEPTH+1);
36          #pragma omp task
37          JoEmQuedoACasa(G+n4*3, H+n4*3, D+n4*3, n-n4*3, DEPTH+1);
38       }
39       else {
40          JoEmQuedoACasa(G+n4*0, H+n4*0, D+n4*0, n4, DEPTH+1);
41          JoEmQuedoACasa(G+n4*1, H+n4*1, D+n4*1, n4, DEPTH+1);
42          JoEmQuedoACasa(G+n4*2, H+n4*2, D+n4*2, n4, DEPTH+1);
43          JoEmQuedoACasa(G+n4*3, H+n4*3, D+n4*3, n-n4*3, DEPTH+1);
44       }
45    }
46 }
47
48 void main() {
49    ...
50
51    int i;
52
53    for (i=0; i<MAX_MAP; i++)
54       omp_init_lock(&lock_global[i]);
55
56    #pragma omp parallel
57    #pragma omp single
58    JoEmQuedoACasa(G, H, D, (1<<18), 0);
59    ...
60
61    for (i=0; i<MAX_MAP; i++)
62       omp_destroy_lock(&lock_global[i]);
63 }
```

Problem 4: iterative task decomposition (questions 11 to 13) (2 points) In this problem the sequential code fo a code excerpt is given. The code traverses the elements in an input data structure (`list_in`) to produce the elements in an output data structure (`list_out`). Three versions of code are available: two containing a countable loop (for) and another one containing an uncountable loop (while). The two with a countable loop only differ in the way a new element is inserted in the output vector.

Version 1: countable loop

```
// Note: The order of the elements in list_out is not relevant,
// that is, they can be inserted in different order than in sequential.
// The type of the elements of the lists can be ANY TYPE.
int n_in_elems = 1000;
int n_out_elems = 0;
int i;

for (i=0 ; i<n_in_elems; i++)
    if (FUNCTION_NAME(list_in[i]) != 0 ) {
        list_out[n_out_elems] = PROCESS(list_in[i]);
        n_out_elems++;
    }
}
```

Version 2: countable loop with temporary variable

```
// Note: The order of the elements in list_out is not relevant,
// that is, they can be inserted in different order than in sequential.
// The type of the elements of the lists can be ANY TYPE.
int n_in_elems = 1000;
int n_out_elems = 0;
int i, position;

for (i=0 ; i<n_in_elems; i++)
    if (FUNCTION_NAME(list_in[i]) != 0 ) {
        position = n_out_elems;
        n_out_elems++;
        list_out[position] = PROCESS(list_in[i]);
    }
}
```

Version 3: uncountable loop

```
// Note: The order of the elements in list_out is not relevant,
// that is, they can be inserted in different order than in sequential.
// The type of the elements of the lists can be ANY TYPE.
int position, n_out_elems = 0; // and list_in previously initialized

while (list_in != NULL) {
    if (FUNCTION_NAME(list_in->element) != 0) {
        position = n_out_elems;
        list_out[position] = PROCESS(list_in->element);
        n_out_elems++;
    }
    list_in = list_in->next;
}
```

Functions `FUNCTION_NAME` and `PROCESS` are computationally expensive functions with either a constant execution time or with significantly variable execution time (depending on the arguments). In any case, none of these functions modify any memory contents. Finally, the problem restricts the use of certain OpenMP constructs to write the parallel code:

- R1: You are ONLY allowed to USE EXPLICIT TASKS, but NOT allowed to USE TASKLOOP, making explicit the data sharing clauses for the variables that are relevant.
- R2: You are ONLY allowed to USE EXPLICIT TASKS generated with TASKLOOP, making explicit the data sharing clauses for the variables that are relevant.
- R3: you are ONLY allowed to USE IMPLICIT TASKS WITHOUT USING FOR worksharing.
- R4: you are ONLY allowed to USE IMPLICIT TASKS AND FOR worksharing.

In all cases you are allowed to use PARALLEL and SINGLE when necessary, and any clause that is available for the allowed OpenMP constructs. For code version 3 (uncountable loop) restriction R1 always applied.

Question 11: Explain the most appropriate distribution of the iterations of the loop to tasks for this code, briefly justifying your answer in terms of parallelization limitations, overheads and inefficiencies.

In all versions of the code in this problem there is always a potential problem of load balancing, caused in the first place by the fact that some iterations enter the conditional statement and others do not; and

secondly because the function PROCESS, invoked from within the conditional, may have a variable execution time or not. Based on that, iterations should be preferably distributed among implicit or explicit tasks in a dynamic way in order to dynamically balance the load assigned to each processor. This is easy to achieve except when the problem applies restriction R3.

When restrictions R1 or R2 apply, it is important to avoid distributions that only generate a number of tasks equal to the number of processors to be used in the parallel execution; but also since there are a very large number of iterations in the loop, it is important to avoid the other extreme case of 1 iteration per task. Determining the number of consecutive iterations per task is beyond the scope of the problem, but it should be identified that this should be done. In any case, with explicit tasks and `taskloop` (R2) the use of `grainsize` or `num_tasks` is needed to control task granularity; with explicit tasks without being able to use `taskloop` (R1) the solution involves the use of loop blocking (i.e. divide the original loop in two so that the outer loop traverses chunks and the inner loop traverses the iterations within each chunk).

When restrictions R3 or R4 apply, then the number of tasks (implicit) is fixed to the number of processors. With implicit tasks and `for` (R4) it is possible to play with the `chunk` size of the `dynamic` schedule or with a `guided` schedule that dynamically adjusts the chunk size. With implicit tasks without being able to use `for`, the implementation limits us to a static (manual) distribution of iterations based on `omp_get_thread_num()` and `omp_get_num_threads()`; it would be possible to implement dynamic schemes but this was not the purpose for this problem.

For version 3 of the code (the one with while loop), since the number of iterations to be performed is not known, the simplest solution is to perform a task for each iteration, all generated by the implicit task that enters a `single` worksharing construct.

Question 12: Which is the potential concurrency problem that may happen when iterations of the loop body are executed in parallel? Which variables and accesses to them are creating these concurrency problems? Briefly justify your answer.

In this code there are potential data races caused by the access to shared variables that are read and written in the loop iterations. In particular, variable `n_out_elems` has to be protected so that both the reading of `n_out_elems` and its subsequent increase and write in `n_out_elems ++` occurs atomically. If temporary variable `position` is used in your code version, that variable should be privatized to avoid data races on it. Both things ensure that each task writes to different positions in the `list_out` vector. The access to `list_in` is only for reading, so no data races are possible; we will only need to ensure each task gets the appropriate element (or elements) to process.

It is important to observe that with atomic it is not possible to protect both instructions simultaneously, so either `critical` or locks should be used. It is also important that function PROCESS is invoked outside the critical region to avoid its serialization and the corresponding loss of a significant parallel fraction. It is not possible to use reduction because it privatizes variable `n_out_elems`, initializing the private copy to 0 and making all tasks/threads write on the same `list_out` positions.

Question 13: Write an OpenMP parallel code, ONLY using the allowed OpenMP constructs, that implements an iterative linear task decomposition strategy. Your parallel implementation should be coherent with the reasoning you used in the two questions above, and whenever possible, reduce the sections of code that are executed sequentially or are serialised due to the use of synchronization constructs. Briefly explain the decisions you took when writing the parallel code.

The different solutions are given below. The difference between Versions 1 and 2 is only shown for R1, for the rest of restrictions, and for brevity reasons, only code for version 2 is given; the reader can extrapolate the solution for version 1 based on the differences shown for R1.

Version 1: countable loop with R1

```
int n_in_elems = 1000;
int n_out_elems = 0;
int i;
#define GRAINSIZE ...

#pragma omp parallel
#pragma omp single
{
for (i=0 ; i<n_in_elems; i+=GRAINSIZE)
    #pragma omp task firstprivate(i) shared(list_in, list_out, n_out_elems)
    for (int ii=i ; ii<i+GRAINSIZE; ii++)              // minimum with n_in_elems should be included
        if (FUNCTION_NAME(list_in[ii]) != 0) {
            TYPE tmp = PROCESS(list_in[ii]);
            #pragma omp critical
            {
            list_out[n_out_elems] = tmp;
            n_out_elems++;
            }
        }
}
```

Version 2: countable loop with temporary variable with R1

```
int n_in_elems = 1000;
int n_out_elems = 0;
int i, position;
#define BS ...

#pragma omp parallel
#pragma omp single
{
for (i=0 ; i<n_in_elems; i+=BS)
    #pragma omp task private(position) firstprivate(i) shared(list_in, list_out, n_out_elems)
    for (int ii=i ; ii<i+BS; ii++)                     // minimum with n_in_elems should be included
        if (FUNCTION_NAME(list_in[ii]) != 0) {
            #pragma omp critical
            {
            position = n_out_elems;
            n_out_elems++;
            }
            list_out[position] = PROCESS(list_in[ii]);
        }
}
```

Version 1: countable loop with R2 (omitted for brevity)
Version 2: countable loop with temporary variable with R2

```
int n_in_elems = 1000;
int n_out_elems = 0;
int i, position;

#pragma omp parallel
#pragma omp single
{
#pragma omp taskloop grainsize(BS) private(i, position) shared(list_in, list_out, n_out_elems)
for (i=0 ; i<n_in_elems; i++)
    if (FUNCTION_NAME(list_in[i]) != 0) {
        #pragma omp critical
        {
        position = n_out_elems;
        n_out_elems++;
        }
        list_out[position] = PROCESS(list_in[i]);
    }
}
```

Version 1: countable loop with R3 (omitted for brevity)
Version 2: countable loop with temporary variable with R3

```
int n_in_elems = 1000;
int n_out_elems = 0;
int i, position;

#pragma omp parallel private(i, position) shared(list_in, list_out, n_out_elems)
{
int howmany = omp_get_num_threads();
int whoami = omp_get_thread_num();

for (i=whoami ; i<n_in_elems; i+=howmany)
    if (FUNCTION_NAME(list_in[i]) != 0) {
        #pragma omp critical
        {
        position = n_out_elems;
        n_out_elems++;
        }
        list_out[position] = PROCESS(list_in[i]);
    }
}
```

Version 1: countable loop with R4 (omitted for brevity)

Version 2: countable loop with temporary variable with R4

```
int n_in_elems = 1000;
int n_out_elems = 0;
int i, position;
#define BS ...
                                                // or guided but not static
#pragma omp parallel for private(position) schedule(dynamic, BS) shared(list_in, list_out, n_out_elems)
for (i=0 ; i<n_in_elems; i++)
    if (FUNCTION_NAME(list_in[i]) != 0) {
        #pragma omp critical
        {
        position = n_out_elems;
        n_out_elems++;
        }
        list_out[position] = PROCESS(list_in[i]);
    }
```

Version 3: uncountable loop with R1

```
int position, n_out_elems = 0; // and list_in previously initialized

#pragma omp parallel
#pragma omp single
while (list_in != NULL) {
    #pragma omp task firstprivate(list_in) private(position) shared(list_out, n_out_elems)
    if (FUNCTION_NAME(list_in->element) != 0) {
        #pragma omp critical
        {
        position = n_out_elems;
        n_out_elems++;
        }
        list_out[position] = PROCESS(list_in->element);
    }
    list_in = list_in->next;
}
```