University of Ioannina

Department of Computer & Informatics Engineering

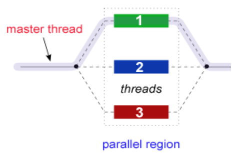
Parallel Systems & Programming

Spring Semester 2024

OpenMP

Christos Dimitresis 4351
cs04351@uoi.gr

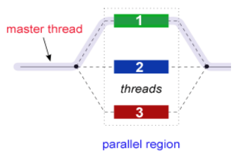
Contents



Measurement Environment

All experimental timing - performance measurements of the department's computer implementations with the following characteristics.

Computer Name	opti7020ws11
Processor Model	Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
Number of Cores	4
Translator	gcc 11.2.0



Parallelization of Finding Prime Numbers

Data - Required

Given a serial program in which, given N, the function serial_primes() computes the number of primes as well as the largest prime up to N.

The openmp_primes() function is requested to do the same calculations in parallel, using OpenMP.

No change is allowed in the implemented algorithm.

Loop Parallelization

In order to parallelize the given serial program, instructions using the OpenMP API were used.

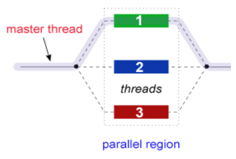
Specifically with the directive :
#pragma omp parallel for private(num, divisor, quotient , remainder) reduction(+ : count) reduction(max : lastprime) schedule(runtime)
we parallelize the first loop we encounter in our serial program.

The variables ***num, divisor, quotient, remainder*** must be private for each thread that will be created because according to the algorithm of the code, they change - they are updated at each iteration of the loop. So we make sure this happens only with the private directive.

All threads will have to update the **initially** shared ***count*** and ***lastprime variables*** , so race conditions are created. To solve this problem we use the directives ***reduction(max : lastprime)*** and ***reduction(+ : count)***
With ***reduction(max : lastprime)*** a separate local variable is created for each thread and at the end of the execution of all the threads, the value of the ***lastprime variable is chosen as*** the numerically largest value of them.

With the directive ***reduction(+ : count)*** a separate local variable count is created in each thread and at the end of the execution of all threads the numerical sum of all these local variables is selected as the value of the ***count variable.***

The implementation to create a critical region ***#pragma omp critical*** and a lock at the point where the value of the ***count variable is updated*** in the code was also tested, but it did not offer any noticeable time improvement compared to ***reduction*** .
The ***schedule(runtime) directive*** allows us to define through the terminal environment variable the thread allocation policy making the timing process more flexible



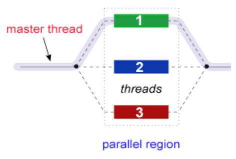
Timing

The programs were executed on [a computer](#)with the technical characteristics mentioned in the introduction and for time measurement the ***gettimeofday() function from the <sys/time.h>*** library was used

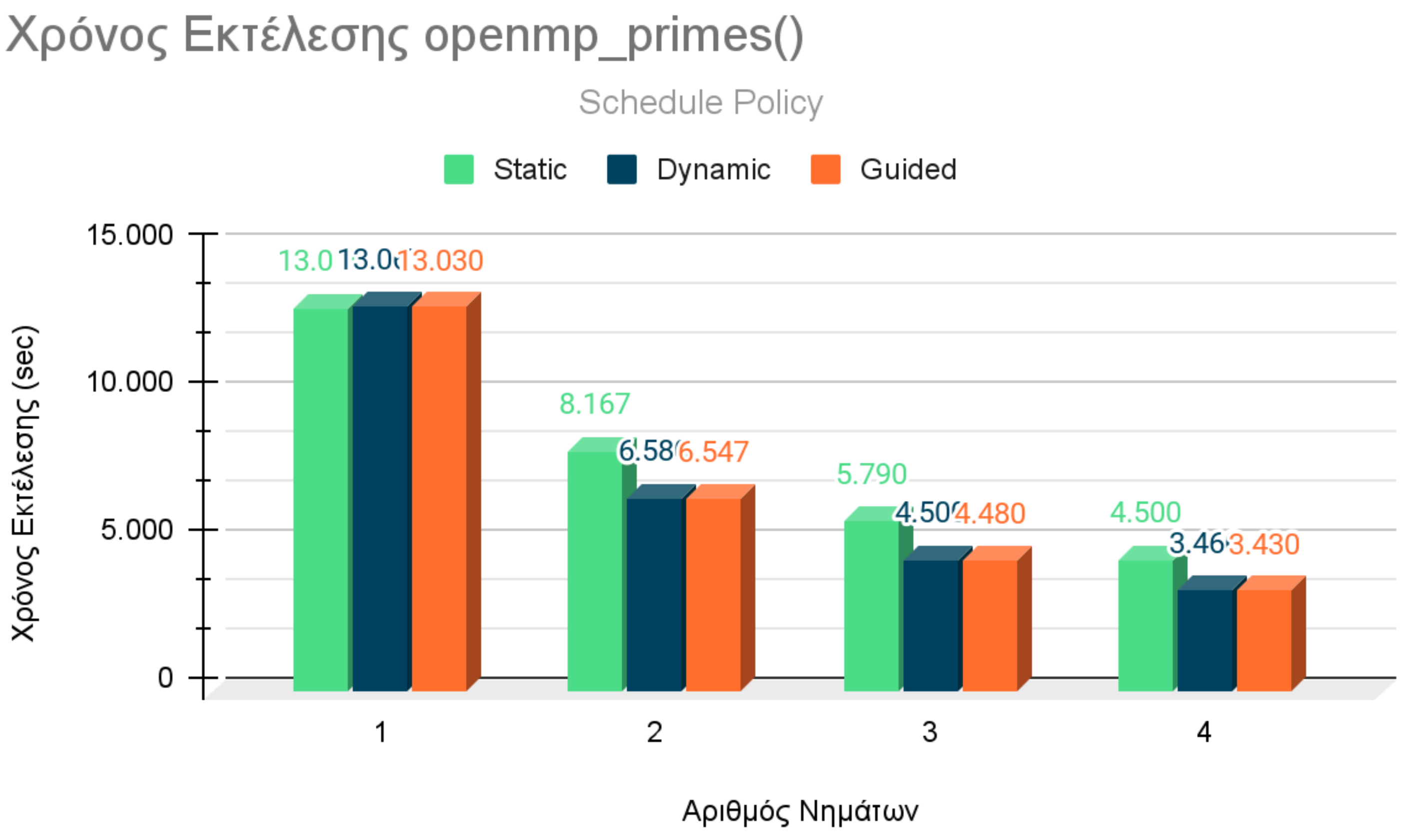
It is emphasized that in each sharing policy the chunk has not been explicitly defined and therefore the one predefined by the system is used.

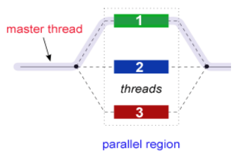
Static Schedule					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	13.01 sec	13.02 sec	13.02 sec	13.01 sec	13.015 sec
2	8.17 sec	8.16 sec	8.17 sec	8.17 sec	8.167 sec
3	5.79 sec	5.79 sec	5.79 sec	5.79 sec	5,790 sec
4	4.50 sec	4.50 sec	4.50 sec	4.50 sec	4,500 sec

Dynamic Schedule					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	13.07 sec	13.07 sec	13.07 sec	13.06 sec	13.067 sec
2	6.58 sec	6.58 sec	6.58 sec	6.58 sec	6,580 sec
3	4.50 sec	4.50 sec	4.50 sec	4.50 sec	4,500 sec
4	3.46 sec	3.46 sec	3.46 sec	3.46 sec	3,460 sec



Guided Schedule					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	13.03 sec	13.03 sec	13.03 sec	13.03 sec	13.030 sec
2	6.55 sec	6.55 sec	6.54 sec	6.55 sec	6,547 sec
3	4.48 sec	4.48 sec	4.48 sec	4.48 sec	4,480 sec
4	3.43 sec	3.43 sec	3.43 sec	3.43 sec	3,430 sec





Conclusion

Studying the graph above, which correlates the number of threads in relation to the allocation policy and the total execution time, the following conclusions are drawn.

Regarding the percentage and also the quality of the generated parallelization of the code, we can claim that we have reached a satisfactory level since in all tables the relation -> Time_with_N_Threads ≈ Time_Serial/N is valid

So the work has been shared equally across all threads and we have reached the maximum possible parallelization.

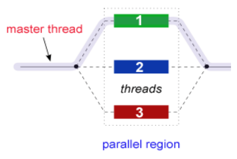
—

Regarding the distribution policy, it is obvious that **static** distribution is the most time-consuming, while **dynamic** and **guided** are equivalent and significantly faster. This happens because of the no-auto-routing policy in **static** .

static policy and not setting chunk_size , the default is set which is to divide the loop iterations into groups of equal number of iterations and give them to the cores. But while the number of repetitions given to each core will be the same, the load will not be. Because as i increases in the loop, the more execution time will be needed by the while contained in each iteration (according to the algorithm implemented by the code). So the load is not shared equally among the threads and as a result this small delay.

By using the **dynamic policy** without having defined **chunk_size,** the default is used, i.e. assigning a repetition to each thread and therefore if any thread is more time-consuming it will not prevent the execution of the others because there is no predetermined order of execution of the threads but competition between them.

guided policy in our timings turns out to be equivalent to the **dynamic one** due to the small number of cores - threads used in the timing. Four (4) threads is not a large enough number in order to create congestion and contention problems between threads. Only then would there be any noticeable benefit to the different way of assigning threads used by the **guided policy** compared to the **dynamic one** .



Blurring of Images

Data - Required

Given a serial program that applies a Gaussian blur in order to blur (or normalize) an image.

The function that does the blurring is gaussian_blur_serial(), which takes an image
imgin and produces its blurred version imgout, based on a blur radius (the larger the radius, the more intense the blur).

It is requested that blurring be done in parallel using OpenMP with parallelization of loops and also with the use of tasks.

Loop Parallelization - Implementation

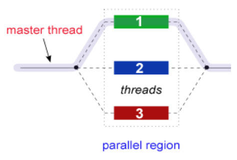
In order to parallelize the given serial program, instructions using the OpenMP API were used.

The function
void gaussian_blur_omp_loops(int radius, img_t *imgin, img_t *imgout)
uses the serial implementation of the function as a basis
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
in order to parallelize the code loops.

Specifically in the function
void gaussian_blur_omp_loops(int radius, img_t *imgin, img_t *imgout)
with the directive
#pragma omp parallel for private(i,j,row,col,redSum, greenSum, blueSum, weightSum) collapse(2) schedule(runtime)
we parallelize the first loop we encounter in our code.

#collapse(2) directive merges the first 2 for-loops into one, which is possible because the first two loops are independent of each other, have regular form, and no code is included between them. This directive increases the parallelism of our program by essentially eliminating 1 of the 4 inner loops.

schedule(runtime) directive allows us to define through the terminal environment variable the thread allocation policy, making the timing process more flexible.



The directive ***private(i,j,row,col,redSum, greenSum, blueSum, weightSum)*** defines the specific local variables for each thread, a necessary fact because in each iteration of the code rains these variables are read - modified.

The directive ***shared(radius,imgin,imgout,width,height)*** defines the specified variables as shared. The variables ***radius, imgin, width, height*** are not modified during the execution of the loop code, they are only read with a fixed value, so there are no race-conditions problems and we don't have a problem with them being shared between threads.

The ***imgout variable*** is updated during the execution of the loops, but according to the implemented algorithm it is guaranteed that each thread will write to a different point of the table pointed to by the **imgout pointer** . So given that, there is no race-condition problem and it is possible for this variable to be shared between threads.

Parallelization using Tasks

In order to parallelize the given serial program, instructions using the OpenMP API were used.

The function
void gaussian_blur_omp_tasks(int radius, img_t *imgin, img_t *imgout)
uses the serial implementation of the function as a basis
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
in order to parallelize the code using tasks.

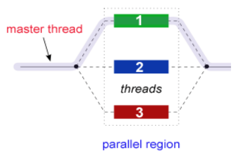
We are right before the first of the 4 loops in the code.

#pragma omp parallel directive creates a parallel region and threads. Within this parallel region the directive ***#pragma omp single*** creates a region in which the code it contains will be executed by only one thread of those created in the parallel region. In our case the parallel region and the single region are identical but it could have been different.

The instruction
#pragma omp task shared(imgin,imgout,radius,width,height) private(row,col) firstprivate(i,j,weightSum,redSum,greenSum,blueSum)
creates the Tasks. All these Tasks that will be created, after the end of the single area and due to the barrier that exists at the end of it, will start to be consumed and executed in parallel by all the threads that were created when we created the parallel area at the beginning.

Regarding the directive that creates the Tasks:

shared(imgin,imgout,radius,width,height) directive defines the specific variables shared between all Tasks.



The variables ***radius, imgin, width, height*** are not modified during the execution of each Task, but are only read with a fixed value, so there are no race-conditions problems and there is no problem that they are shared between Tasks.

The ***imgout variable*** is updated during the execution of the code of each Task, but according to the implemented algorithm it is guaranteed that each Task will write to a different point of the table shown by the ***imgout pointer .*** So given this there is no race-condition problem and it is possible for this variable to be shared between Tasks.

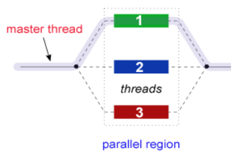
The directive ***private(row,col)*** defines the specific variables and private for each Task. According to the code, these variables are initialized within the respective Task.

The directive ***firstprivate(i,j,weightSum,redSum,greenSum,blueSum)*** defines the specific private variables for each Task. The difference with the private directive lies in the fact that these variables are initialized at the moment of creation of the Task with the values they had at that moment "externally", which is also the desired thing because they are also used in code outside the Task, so they change over time. We do not know when each Task will be executed, so we ensure that when it is executed, its parameters will have the values they had at the time of its creation.

Timing

The programs were executed on a computer with the technical characteristics mentioned in the introduction and for time measurement the ***gettimeofday() function from the <sys/time.h>*** library was used

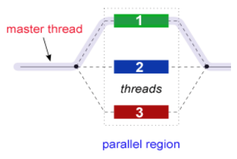
It is emphasized that in each sharing policy the chunk has not been explicitly defined and therefore the one predefined by the system is used.



Static Schedule - Parallel Loop					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	18.36 sec	18.31 sec	18.31 sec	18.30 sec	18,320 sec
2	9.21 sec	9.21 sec	9.22 sec	9.21 sec	9,212 sec
3	6.30 sec	6.30 sec	6.30 sec	6.30 sec	6,300 sec
4	4.90 sec	4.86 sec	4.86 sec	4.86 sec	4,870 sec

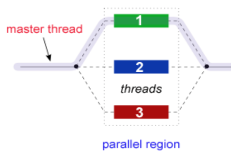
Dynamic Schedule - Parallel Loop					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	18.36 sec	18.35 sec	18.35 sec	18.36 sec	18.357 sec
2	9.31 sec	9.29 sec	9.30 sec	9.31 sec	9,302 sec
3	6.37 sec	6.37 sec	6.37 sec	6.37 sec	6,370 sec
4	4.90 sec	4.91 sec	4.90 sec	4.91 sec	4.905 sec

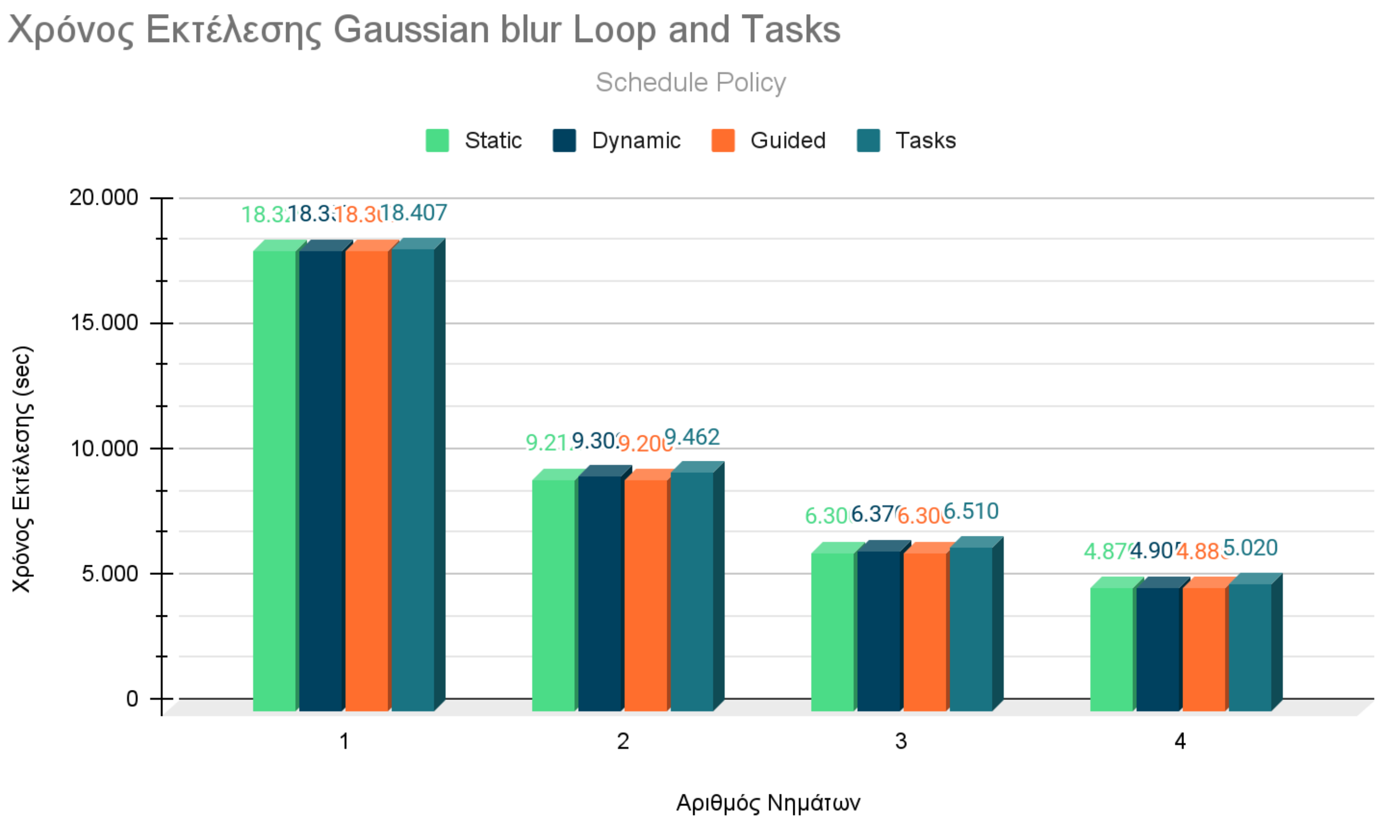
Guided Schedule - Parallel Loop					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	18.30 sec	18.31 sec	18.31 sec	18.31 sec	18.307 sec



2	9.17 sec	9.21 sec	9.21 sec	9.21 sec	9,200 sec
3	6.30 sec	6.30 sec	6.30 sec	6.30 sec	6,300 sec
4	4.86 sec	4.92 sec	4.88 sec	4.88 sec	4,885 sec

Parallel Tasks					
Threads	1st execution	2nd execution	3rd execution	4th execution	Average
1	18.40 sec	18.42 sec	18.41 sec	18.40 sec	18.407 sec
2	9.47 sec	9.46 sec	9.46 sec	9.46 sec	9,462 sec
3	6.51 sec	6.51 sec	6.52 sec	6.52 sec	6,510 sec
4	5.02 sec	5.02 sec	5.02 sec	5.02 sec	5,020 sec





Conclusion

Studying the above graph which correlates the number of threads in relation to the allocation policy and the total execution time, the following conclusions are drawn.

Regarding the percentage and also the quality of the generated parallelization of the code, we can claim that we have reached a satisfactory level since in all tables the relation -> Time_with_N_Threads ≈ Time_Serial/N is valid

So the work has been shared equally across all threads and we have reached the maximum possible parallelization.

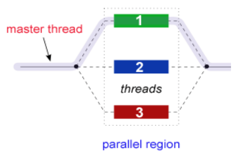
Regarding the sharing policy it is obvious that all policies are equivalent in time and there is no benefit to using one over another.

Using Tasks could abstractly be thought of as a sharing policy similar to using dynamic with chunk = 1 since our code is not random and essentially uses loops again.

So all the above allocation policies in the case of loop parallelization end up equivalent due to the fact that the load in each iteration of the (outer) loop is overall equivalent and also the iterations are independent.

In the case of Tasks, exactly the same applies. Each Task has an equivalent load and is independent of each other.

Given also the relatively large number of nested loops (4) that we have in the version of the parallelization of the loop, the time burden for the creation of the Tasks in the version with the Tasks and finally all the aforementioned ways of executing the code and also of sharing the load is compensated, they end up being equivalent.



Task Dependencies

Task Dependencies is essentially a Task synchronization feature available from OpenMP version 4.0 onwards. With the use of these instructions we can associate Tasks with each other, increasing the parallelism of our implementation while maintaining a form of pipeline in order not to lose the association of the Task data.

In order to make their operation more understandable we will implement a trivial case of needing synchronization using Task Dependencies. This case is the known to all of us "producer-consumer" synchronization problem

In our implementation we read a txt text file and as we read it we want to print its content to the terminal. We will use Tasks in order to parallelize the said process and Task Dependencies to synchronize the Tasks and the printing result to be in the correct order. That is, the contents of the file should be printed in the order they were read and not randomly.

Here is the code that implements our example and the input file:

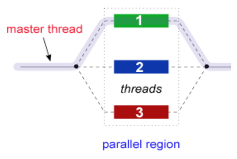
```
1 non-host device has a unique device number that is greater than or equal to zero and less than the
2 device number for the host device. Additionally, the constant omp_initial_device can be
3 used as an alias for the host device and the constant omp_invalid_device can be used to
4 specify an invalid device number. A conforming device number is either a non-negative integer that
5 is less than or equal to omp_get_num_devices() or equal to omp_initial_device or
6 omp_invalid_device.
7 When a target construct is encountered, a new target task is generated. The target task region
8 encloses the target region. The target task is complete after the execution of the target region
9 is complete.
10 When a target task executes, the enclosed target region is executed by an initial thread. The
11 initial thread executes sequentially, as if the target region is part of an initial task region that is
12 generated by an implicit parallel region. The initial thread may execute on the requested target
13 device, if it is available and supported. If the target device does not exist or the implementation
14 does not support it, all target regions associated with that device execute on the host device.
15 The implementation must ensure that the target region executes as if it were executed in the data
16 environment of the target device unless an if clause is present and the if clause expression
17 evaluates to false.
18 The teams construct creates a league of teams, where each team is an initial team that comprises
19 an initial thread that executes the teams region. Each initial thread executes sequentially, as if the
20 code encountered is part of an initial task region that is generated by an implicit parallel region
21 associated with each team. Whether the initial threads concurrently execute the teams region is
22 unspecified, and a program that relies on their concurrent execution for the purposes of
23 synchronization may deadlock.
24 If a construct creates a data environment, the data environment is created at the time the construct is
25 encountered. The description of a construct defines whether it creates a data environment.
26 When any thread encounters a parallel construct, the thread creates a team of itself and zero or
27 more additional threads and becomes the primary thread of the new team. A set of implicit tasks,
28 one per thread, is generated. The code for each task is defined by the code inside the parallel
29 construct. Each task is assigned to a different thread in the team and becomes tied? that is, it is
30 always executed by the thread to which it is initially assigned. The task region of the task being
31 executed by the encountering thread is suspended, and each member of the new team executes it
32 implicit tasks. An implicit barrier occurs at the end of the parallel region. Only the primary
33 thread resumes execution beyond the end of the parallel construct, resuming the task region
34 that was suspended upon encountering the parallel construct. Any number of parallels
35 constructs can be specified in a single program.
36 parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or
37 is not supported by the OpenMP implementation, then the new team that is created by a thread that
38 encounters a parallel construct inside a parallel region will consist only of the
39 encountering thread. However, if nested parallelism is supported and enabled, then the new team
```

Input File :
openMP_execution_model.txt

⇐

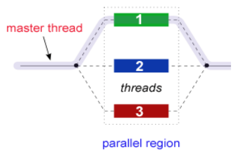
*** At the beginning of each line there is for convenience the number that corresponds to it so that the printing on the console is visible if it was done keeping the correct order of the file.

Program Code : task_dependencies.c ⤵



```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <omp.h>

#define BUFFER_SIZE 50

char buffer [BUFFER_SIZE];
int fp ;
long file_size ;

void reader ( size_t nbytes ){
   read ( fp , buffer , nbytes );
}

void writer ( size_t nbytes ){
   write (STDOUT_FILENO, buffer , nbytes );
}

int main ( void ) {

   fp = open ( "openMP_execution_model.txt" , O_RDONLY);

   if ( fp == - 1 ) {
       perror ( "Opening file failed. System exit" );
       return - 1 ;
}

   file_size = lseek ( fp , 0 , SEEK_END);
   lseek ( fp , 0 , SEEK_SET);

   int iterations_num = file_size / BUFFER_SIZE;
   size_t last_chunk_bytes = file_size % BUFFER_SIZE;

   #pragma omp parallel
   #pragma omp single
{
       for ( int i = 0 ; i <= iterations_num ; i ++ ) {
           #pragma omp task depend(out : buffer)
           writer (BUFFER_SIZE);

           #pragma omp task depend(in : buffer)
           reader (BUFFER_SIZE);
}
}

   reader ( last_chunk_bytes );
   writer ( last_chunk_bytes );

}
```

Code Explanation



1. We open the file we mentioned before in order to read it later.

2. We calculate its size in bytes and divide it by the size of the buffer we have in order to calculate how many Tasks writer and reader it is necessary to create afterwards. If the size of the txt file in bytes is not exactly divided by the size of the buffer, the rest of the remaining bytes are read and written serially after the end of the parallel area.

3. We create a parallel region. This is also where the threads are created.

4. **single** region whose code will only be executed by one of the previously created threads.

   single area we create the **Tasks** of our algorithm. By using the loop we numerically create as many Tasks as are necessary based on the logic explained in step 2.

   In each iteration, a **Task is created** to read **buffer_size** bytes of the archive and also a **Task** to print to the terminal **buffer_size** bytes from the buffer.

   Tasks that contain the functions **writer()** and **reader()** respectively.

   These **Tasks** are not executed at the time of their creation (because we are inside the **single area**), but they will start to be executed as soon as we leave the **single area** and "fall" on the **barrier** that exists at the end of this area. (Which also happens to coincide with the end of the parallel region as a whole).

   So while we have successfully created the necessary number **of Tasks** to read all the bytes of the txt file and print them on the terminal, if we did not use task dependencies there would be a synchronization issue.

   So at the point where we create the tasks after the directive ***#pragma omp task*** which creates a **Task** according to the known way, we add the directive ***depend(out : buffer)*** to the **Tasks** that implement **writer()** and ***depend(in : buffer)*** in **Tasks** that implement **reader()** .

   With the directive ***depend(out : buffer)*** we declare to the system that the **Task** that contains it will write to the **buffer variable** .

   With the directive ***depend(in : buffer)*** we declare to the system that the **Task** that contains it will read the **buffer variable** .

   So based on this information the system can understand the correlation that the Tasks have with each other and that they share a common variable.

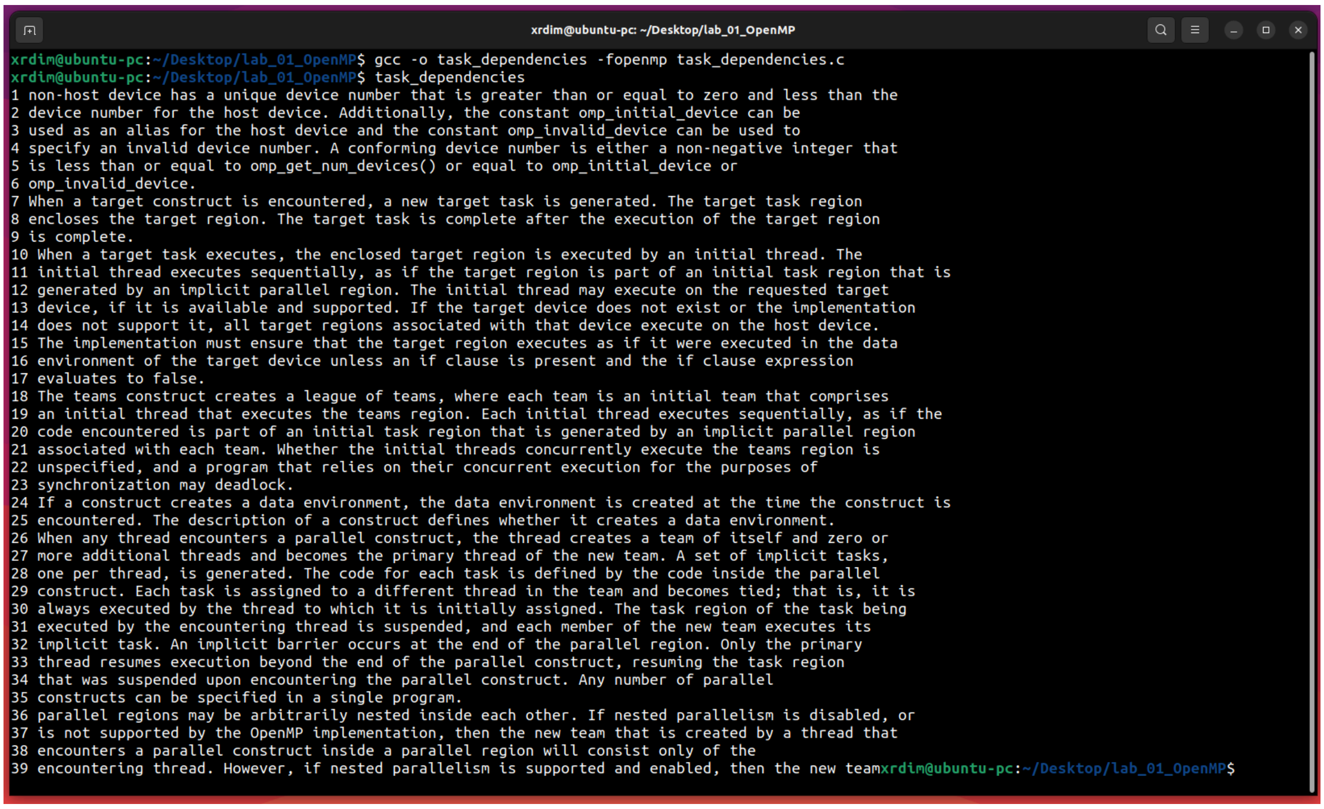
   Tasks will be associated , a **reader()** and a **writer() , and first the Task** that writes (writer) and then the **Task** that reads (reader) the variable **buffer** must be executed . Due to the imposition of this in the execution order of the Tasks, the race-condition problem that would otherwise exist for the **buffer variable is also eliminated** .

So we create a pipeline-like execution structure. Every time a **Task writer() reads the buffer** variable, a **Task reader() can simultaneously** print to the terminal the value that the previous **Task writer() had read** .

So parallelism of work is practically created.

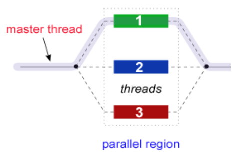
Execution Example

Correct execution using the depend(xxx : buffer) directive

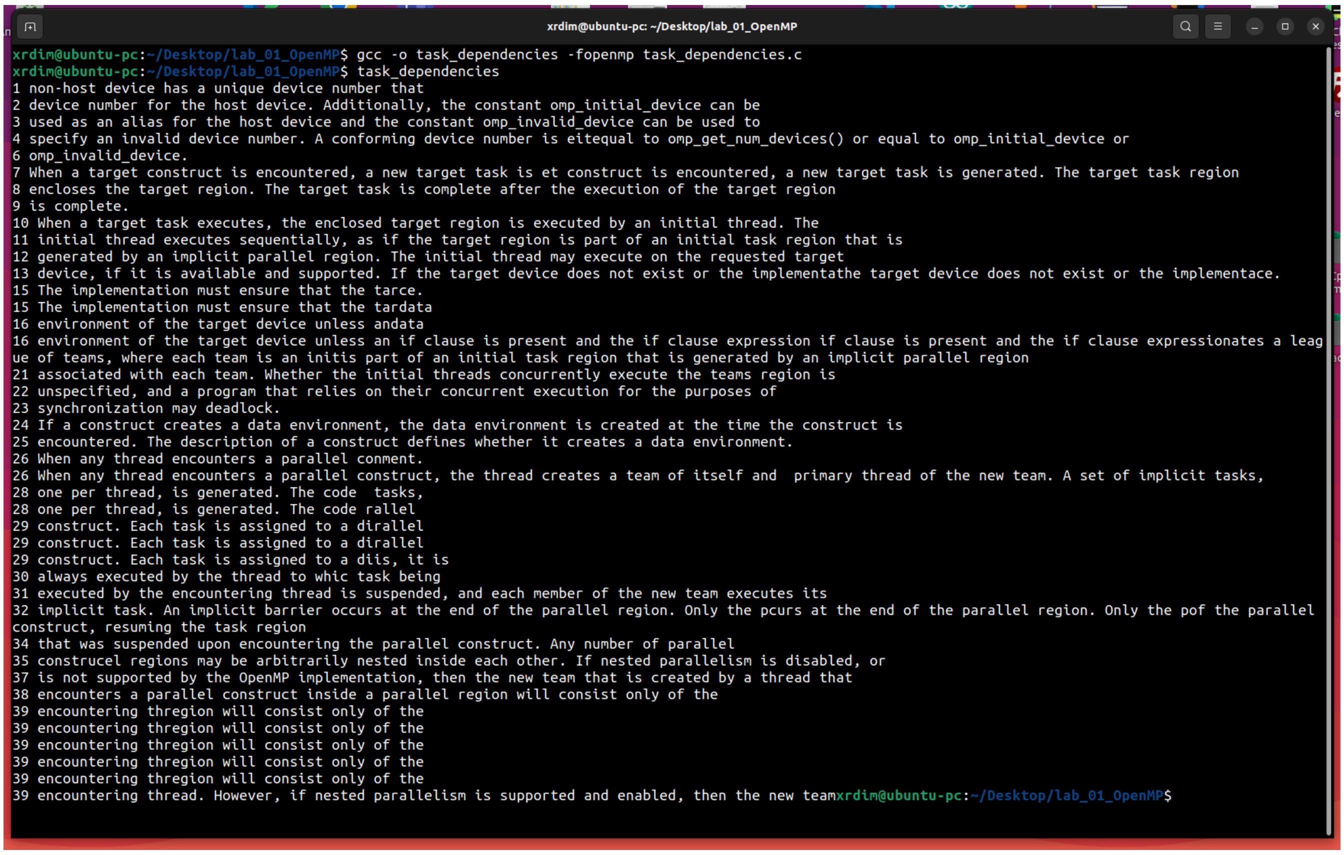


```
xrdim@ubuntu-pc: ~/Desktop/lab_01_OpenMP
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ gcc -o task_dependencies -fopenmp task_dependencies.c
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ task_dependencies
1 non-host device has a unique device number that is greater than or equal to zero and less than the
2 device number for the host device. Additionally, the constant omp_initial_device can be
3 used as an alias for the host device and the constant omp_invalid_device can be used to
4 specify an invalid device number. A conforming device number is either a non-negative integer that
5 is less than or equal to omp_get_num_devices() or equal to omp_initial_device or
6 omp_invalid_device.
7 When a target construct is encountered, a new target task is generated. The target task region
8 encloses the target region. The target task is complete after the execution of the target region
9 is complete.
10 When a target task executes, the enclosed target region is executed by an initial thread. The
11 initial thread executes sequentially, as if the target region is part of an initial task region that is
12 generated by an implicit parallel region. The initial thread may execute on the requested target
13 device, if it is available and supported. If the target device does not exist or the implementation
14 does not support it, all target regions associated with that device execute on the host device.
15 The implementation must ensure that the target region executes as if it were executed in the data
16 environment of the target device unless an if clause is present and the if clause expression
17 evaluates to false.
18 The teams construct creates a league of teams, where each team is an initial team that comprises
19 an initial thread that executes the teams region. Each initial thread executes sequentially, as if the
20 code encountered is part of an initial task region that is generated by an implicit parallel region
21 associated with each team. Whether the initial threads concurrently execute the teams region is
22 unspecified, and a program that relies on their concurrent execution for the purposes of
23 synchronization may deadlock.
24 If a construct creates a data environment, the data environment is created at the time the construct is
25 encountered. The description of a construct defines whether it creates a data environment.
26 When any thread encounters a parallel construct, the thread creates a team of itself and zero or
27 more additional threads and becomes the primary thread of the new team. A set of implicit tasks,
28 one per thread, is generated. The code for each task is defined by the code inside the parallel
29 construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is
30 always executed by the thread to which it is initially assigned. The task region of the task being
31 executed by the encountering thread is suspended, and each member of the new team executes its
32 implicit task. An implicit barrier occurs at the end of the parallel region. Only the primary
33 thread resumes execution beyond the end of the parallel construct, resuming the task region
34 that was suspended upon encountering the parallel construct. Any number of parallel
35 constructs can be specified in a single program.
36 parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or
37 is not supported by the OpenMP implementation, then the new team that is created by a thread that
38 encounters a parallel construct inside a parallel region will consist only of the
39 encountering thread. However, if nested parallelism is supported and enabled, then the new teamxrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$
```

Incorrect execution without using the depend(xxx : buffer) directive



Random Example



xrdim@ubuntu-pc: ~/Desktop/lab_01_OpenMP

```
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ gcc -o task_dependencies -fopenmp task_dependencies.c
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ task_dependencies
1 non-host device has a unique device number that
2 device number for the host device. Additionally, the constant omp_initial_device can be
3 used as an alias for the host device and the constant omp_invalid_device can be used to
4 specify an invalid device number. A conforming device number is eitequal to omp_get_num_devices() or equal to omp_initial_device or
6 omp_invalid_device.
7 When a target construct is encountered, a new target task is et construct is encountered, a new target task is generated. The target task region
8 encloses the target region. The target task is complete after the execution of the target region
9 is complete.
10 When a target task executes, the enclosed target region is executed by an initial thread. The
11 initial thread executes sequentially, as if the target region is part of an initial task region that is
12 generated by an implicit parallel region. The initial thread may execute on the requested target
13 device, if it is available and supported. If the target device does not exist or the implementathe target device does not exist or the implementace.
15 The implementation must ensure that the tarce.
15 The implementation must ensure that the tardata
16 environment of the target device unless andata
16 environment of the target device unless an if clause is present and the if clause expression if clause is present and the if clause expressionates a leag
ue of teams, where each team is an initis part of an initial task region that is generated by an implicit parallel region
21 associated with each team. Whether the initial threads concurrently execute the teams region is
22 unspecified, and a program that relies on their concurrent execution for the purposes of
23 synchronization may deadlock.
24 If a construct creates a data environment, the data environment is created at the time the construct is
25 encountered. The description of a construct defines whether it creates a data environment.
26 When any thread encounters a parallel conment.
26 When any thread encounters a parallel construct, the thread creates a team of itself and  primary thread of the new team. A set of implicit tasks,
28 one per thread, is generated. The code  tasks,
28 one per thread, is generated. The code rallel
29 construct. Each task is assigned to a dirallel
29 construct. Each task is assigned to a dirallel
29 construct. Each task is assigned to a diis, it is
30 always executed by the thread to whic task being
31 executed by the encountering thread is suspended, and each member of the new team executes its
32 implicit task. An implicit barrier occurs at the end of the parallel region. Only the pcurs at the end of the parallel region. Only the pof the parallel
construct, resuming the task region
34 that was suspended upon encountering the parallel construct. Any number of parallel
35 construcel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or
37 is not supported by the OpenMP implementation, then the new team that is created by a thread that
38 encounters a parallel construct inside a parallel region will consist only of the
39 encountering thregion will consist only of the
39 encountering thregion will consist only of the
39 encountering thregion will consist only of the
39 encountering thregion will consist only of the
39 encountering thregion will consist only of the
39 encountering thread. However, if nested parallelism is supported and enabled, then the new teamxrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$
```