

# UNIVERSITAT DE BARCELONA

PARALLEL PROGRAMMING

4TH YEAR

---

## Assignment 1

---

*Delivered to*  
*Prof. de teoria:*  
Paula Gómez

*Delivered by:*  
Alberto Barragán Verdejo 20222370  
David López Adell 20150524

## Contents

<b>1</b>	<b>What is OpenMP?</b>	<b>1</b>
<b>2</b>	<b>First run</b>	<b>1</b>
<b>3</b>	<b>Using more flags</b>	<b>2</b>
3.1	Different flags . . . . .	2
3.2	Results . . . . .	3
<b>4</b>	<b>Create a convertGRB2RGBA_2 function</b>	<b>3</b>
4.1	Results . . . . .	4
<b>5</b>	<b>Keeping the optimizations</b>	<b>4</b>
5.1	Which problems may “struct uchar3” pose regarding memory alignment? .	4
5.2	What is __attribute__ and what does aligned(4) do? . . . . .	5
<b>6</b>	<b>New convertGRB2RGBA_3</b>	<b>5</b>
6.1	Which for loop is better to parallelize? . . . . .	5
6.2	Scheduling types . . . . .	5
<b>7</b>	<b>Parallelize EXPERIMENT_ITERATIONS loop</b>	<b>7</b>
<b>8</b>	<b>Printing each thread</b>	<b>8</b>

## 1 What is OpenMP?

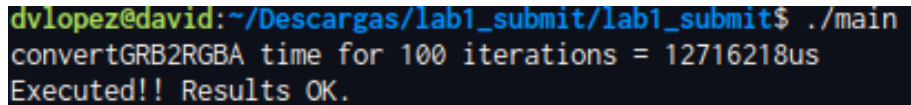
OpenMP, or Open Multi-Processing, is an API that has been used since the late 90s. The API started as a Fortran API in 1997 but later evolved to support C/C++. Ever since the paradigm of the API and Multi-Processing has changed, going from parallelizing regular loops where we know the number of iterations for each loop to a concept of tasks where this wasn't a limitation anymore.

Following this short explanation, we have to remind that OpenMP is not a programming language, as it is an API that manages all the calls to manage the threads and the parallelization of a piece of code.

Nowadays it's mostly used to parallelize for loops in C++ code, mainly in computer science algorithms like merge-sort algorithm. However, it can also be used for function-level parallelism with OpenMP sections, that is another mechanism of OpenMP. This process helps us to improve the execution times of different algorithms in exchange of increasing compilation time, as it will be discussed in more detail in section 3.

## 2 First run

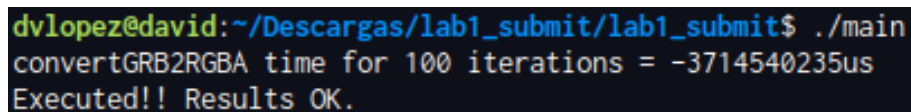
The first thing that we are going to do with the code provided in the assignment, is to run it without any modification in order to have a baseline. Comparing the future results with this baseline, we will be able to notice the changes in the execution times of the code that we are working with.



```
dvlopez@david:~/Descargas/lab1_submit/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = 12716218us
Executed!! Results OK.
```

Figure 1: First execution of the code

However, we ran into a problem executing this the first time. We got a negative result!



```
dvlopez@david:~/Descargas/lab1_submit/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = -3714540235us
Executed!! Results OK.
```

Figure 2: Execution of the code with the negative result that got discarded.

Moreover, after running the code a pair of times until we got some stable results, we understood that the code is converting an image from an *grb* space to an *rgba* space making use of a loop function in the `convertGRB2RGBA` method.

## 3 Using more flags

### 3.1 Different flags

All four flags control different optimizations. When we compile the code without any of these flags the compiler will try to make its best to reduce the compilation time and make debugging produce the results that we expect.

When these flags are activated the compiler will change its goal to attempt to improve performance at the expense of compilation time, and likely the ability to debug the code.

Going a bit more into specifics:

- **-O:** It's equivalent to **-O1**. Here the compiler will take a bit more time, and more memory for larger functions. With this flag activated, the compiler will try to reduce the code size and execution time without increasing too much compilation time. The compiler is able to do all of this by activating a series of flags like:
  - fauto-inc-dec
  - fbranch-count-reg
  - fcombine-stack-adjustments
  - fcompare-elim

- **-O2:** Gets the optimization that we've previously done with **-O1** and adds more optimization flags such as:
  - falign-functions
  - fcaller-saves
  - findirect-inlining

We are also using more memory for the preprocessing. It's also recommended optimization level if there are no special needs of the system.

- **-O3:** On this level we also build up from the previous levels, this time we'll activate more optimization flags but this time less. Some of the flags that are activated are the following ones:
  - fipa-cp-clone
  - floopt-interchange
  - floopt-unroll-and-jam

On this optimization level we are vectoring the loops, using CPU registries as AVX and YMM.

- **-Ofast:** On this level we are disregarding all strict standards. We are enabling all the flags that **-O3** enable and we add two more flags(-ffast-math, -fallow-store-data-races), without counting the FORTRAN specific ones). It will enable optimizations that are not valid for all standard-compliant programs too. And, contrary to what we could expect, we're turning off one flag, -fsemantic-interposition.

This level, together with -O3, are less recommended than the other as they're the most memory hungry levels.

However optimizing with this flags doesn't come without risks. As an example, the GNU GCC guide warns us about invoking -O2 on programs that use computed gotos. But it is said, on multiple stackexchanges and stackoverflow posts, that the flags are safe to use if the code has its bases defined correctly and we understand what each flag will be doing to the compiled code.

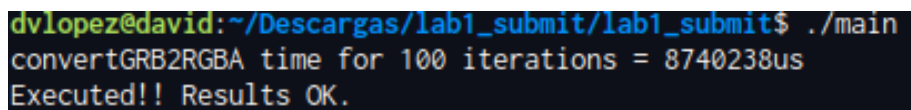
A problem that we found is that if we want to have some exact data in memory, that won't be possible. The optimizer will most likely change it without changing the program behaviour, that means its inputs and outputs, while the internal behaviour will change. But that would be fighting the compiler, a thing that we should always try to avoid when coding.

On another note, we have to take into consideration how much memory requires each of the optimization levels. -O3 and -Ofast can have a big cost as they're pretty memory hungry in comparison to -O1 and -O2. And -O2 is usually the recommended one as it strikes a balance between compilation time and performance. But in the end, knowing our code is the best way to solve this problem, as we will be able to decide which optimization level is better.

We also have to note that if we don't want loop vectoring then the levels -O3 and -Ofast are highly discouraged, as that's one of the things that they do.

### 3.2 Results

The original result was 12716218 microseconds while the result after using the -Ofast flag is 8740238 microseconds.

A terminal window with a dark background and light-colored text. The prompt is 'dvlopez@david:~/Descargas/lab1\_submit/lab1\_submit\$'. The command entered is './main'. The output shows 'convertGRB2RGBA time for 100 iterations = 8740238us' and 'Executed!! Results OK.'

```
dvlopez@david:~/Descargas/lab1_submit/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = 8740238us
Executed!! Results OK.
```

Figure 3: Execution of the code compile with the -Ofast flag.

As we stated, the -Ofast flag usually improves the execution code at the expense of compilation code.

## 4 Create a convertGRB2RGBA\_2 function

In this section, as well as improving data locality, we used the optimization flag -Ofast. The data locality improvement is related on how the loops where iterating height and width. In this case, by iterating first the first row, then the second and so on, we could obtain an improvement of performance, since the original code iterates through the columns instead. This is inefficient because when you access the second row first column you are skipping

n positions (e.g. if the width is 500, the first row first column is position 0 and second row first column position is 500) making each position distance increase as the width of the image increases. Our proposal changes this so we can access position 0, then position 1 and so on till the end of the image.

On the other hand, there's also the spatial locality that happens when a locality is referenced in an specific moment and then is probable that other localities that are nearby can be referenced pronto.

#### 4.1 Results

```
dvlopez@david:~/Descargas/lab1_submit/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = 1113205us
Executed!! Results OK.
```

Figure 4: Results of creating and using convertGRB2RGBA\_2 function.

Here we can note a reduction of approximately 7 million microseconds during this execution.

## 5 Keeping the optimizations

### 5.1 Which problems may “struct uchar3” pose regarding memory alignment?

The struct uchar3 can pose problems as each uchar is a single byte. That means that uchar3 are 3 bytes. We have to take into consideration that for the memory to be aligned we need the bytes that it takes to be a multiple of 2. If this rule is not followed we will have to make several requests to the memory, and as each request is slower than a CPU cycle, we will be wasting lots of CPU cycles per memory access.

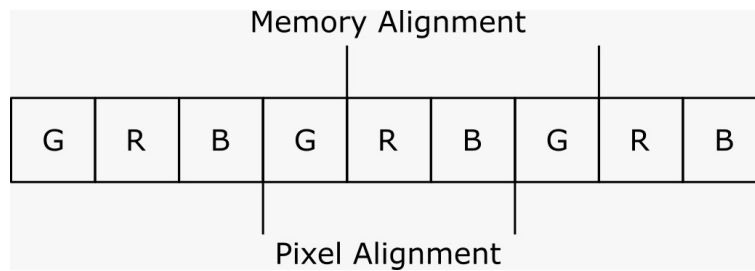


Figure 5: Diagram showing memory alignment with the GRB assignment.

## 5.2 What is `__attribute__` and what does `aligned(4)` do?

Looking in the internet we found out that `__attribute__` is used to specify special properties of variables, like structs. It also gives context and constraints to the compiler so that the compiler can interpret the code in a more efficient way. On the other hand, we have `aligned(int)`. This function establishes a minimum of memory limit, being the `int` the size of memory that we can set. In this case, using a 4, we're forcing it to have a size of 4. And this is an optimum size as it's a power of 2 as it has been explained previously.

## 6 New `convertGRB2RGBA_3`

As the exercise title says, now we have to create a new `convertGRB2RGBA_3` function in the code. We'll play with this function to see which modifications can work better with parallelization in mind.

### 6.1 Which for loop is better to parallelize?

As we can see by the results in the table below, parallelizing the first `for` yields better results than parallelizing the second. However, we could parallelize both `for` loops to have even a further improvement, but that it's not always efficient as the overhead cost of creating more threads could be more expensive than the time that the threads end up reducing.

Loop	Time
1st <i>for</i>	475610us
2nd <i>for</i>	669938us

### 6.2 Scheduling types

The following step of the exercise asks us to test different parameters on different variables to see how the times are changing. We're also changing the scheduling types that as its own name states, are the different schedule plans that the threads can follow to concurrently run loops. Another experimental variable that will come in handy is the `Chunk` size, as it will tell the threads the workload that they will be able to handle. Also, as well as the `chunk` size, we'll be using other variables as the image size or iterations to benchmark how each scheduling type affects our code.

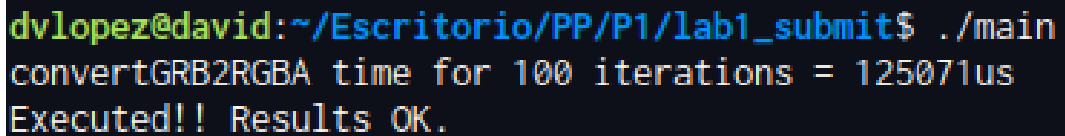
Scheduling type	Chunk size	Width x Height	Experiment iterations	Qualifying times
Dynamic	1	3840 x 2160	100	558511 $\mu$ s
Guided	1	3840 x 2160	100	566157 $\mu$ s
Static	1	3840 x 2160	100	557045 $\mu$ s
Dynamic	50	3840 x 2160	100	558017 $\mu$ s
Guided	50	3840 x 2160	100	573934 $\mu$ s
Static	50	3840 x 2160	100	558212 $\mu$ s
Dynamic	1	7680 x 4320	100	2276203 $\mu$ s
Guided	1	7680 x 4320	100	2256779 $\mu$ s
Static	1	7680 x 4320	100	2251424 $\mu$ s
Dynamic	50	7680 x 4320	100	2305111 $\mu$ s
Guided	50	7680 x 4320	100	2336054 $\mu$ s
Static	50	7680 x 4320	100	2302025 $\mu$ s
Dynamic	1	3840 x 2160	1000	5636415 $\mu$ s
Guided	1	3840 x 2160	1000	5615434 $\mu$ s
Static	1	3840 x 2160	1000	5589721 $\mu$ s
Dynamic	50	3840 x 2160	1000	5585238 $\mu$ s
Guided	50	3840 x 2160	1000	5604593 $\mu$ s
Static	50	3840 x 2160	1000	5590593 $\mu$ s
Dynamic	1	7680 x 4320	1000	23107432 $\mu$ s
Guided	1	7680 x 4320	1000	22481031 $\mu$ s
Static	1	7680 x 4320	1000	22332902 $\mu$ s
Dynamic	50	7680 x 4320	1000	22280554 $\mu$ s
Guided	50	7680 x 4320	1000	22475117 $\mu$ s
Static	50	7680 x 4320	1000	22200139 $\mu$ s



If we see the table we can get the that the average result of the static function is a bit better than the others. However, if we used the scheduling type *auto* we could yield better results, as the strategy is automatically selected. But, as the static strategy was a bit better, we have explored a bit and found that what it does. It is distributing in a uniform and equitable fashion the iterations to the threads using a round-robin algorithm. But, if we increase the size of the image or the number of iterations the computing time will increase accordingly. Making those values higher or lower seemed independent from the *chunk\_size* value. This *chunk\_size* controls the number of iterations by thread and increasing its value makes us feel that the guided scheduling gets closer to the static scheduling, time-wise, as we won't let the number of iterations go below the given threshold by the *chunk\_size* value. Finally, changing the scheduling type doesn't improve the times that much as it seems that they're creating a similar number of threads and there are many calculations being done.

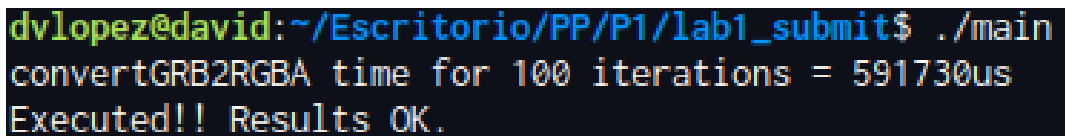
Lastly, regarding the privacy of the variables, we can see that by making all of them private the code won't work as the results from the conversion are wrong. Usually they are shared, that way is how we initially tested the program. We also have to take into account that we want our resources, in this case the *grb* and *rgba* images, to be shared as then we can edit the data being pointed by the pointers. However, we must say that if we make other variables like *width* and *height* as private we get the same error. This error could be because we take a copy of the variables, but the unexpected part is that it doesn't work even if they are never modified, we suspect that when we declare it as random it is being initialized with random values. But it seems that if instead of using private we use *firstprivate* then it works as expected.

## 7 Parallelize EXPERIMENT\_ITERATIONS loop

A terminal window with a dark background and light-colored text. The prompt is 'dvlopez@david:~/Escritorio/PP/P1/lab1\_submit\$'. The command entered is './main convertGRB2RGBA time for 100 iterations = 125071us Executed!! Results OK.'

```
dvlopez@david:~/Escritorio/PP/P1/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = 125071us
Executed!! Results OK.
```

Figure 6: Execution with parallelism

A terminal window with a dark background and light-colored text. The prompt is 'dvlopez@david:~/Escritorio/PP/P1/lab1\_submit\$'. The command entered is './main convertGRB2RGBA time for 100 iterations = 591730us Executed!! Results OK.'

```
dvlopez@david:~/Escritorio/PP/P1/lab1_submit$ ./main
convertGRB2RGBA time for 100 iterations = 591730us
Executed!! Results OK.
```

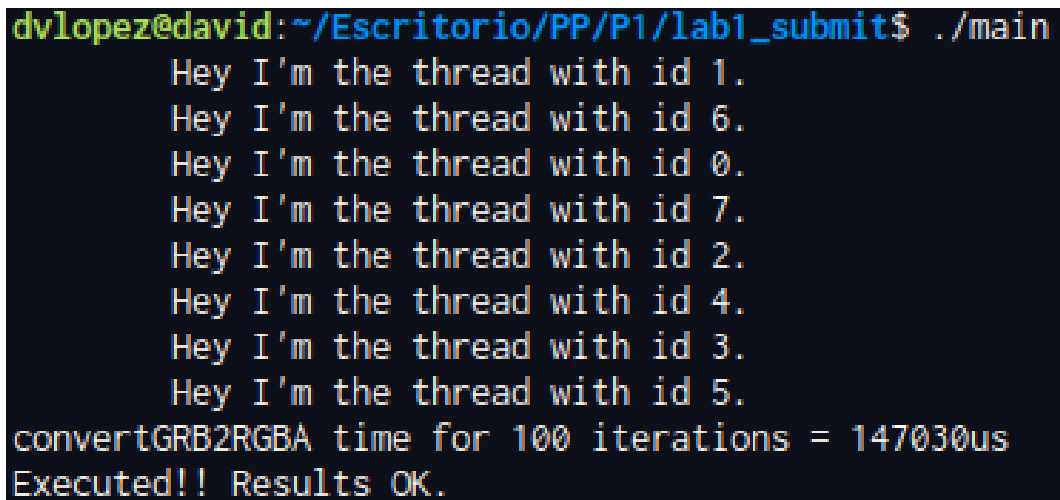
Figure 7: Execution without parallelism

We see that we can obtain a better performance when we parallelize the execution of the conversion function before parallelizing the for loops of the function. This could be

because of the number of threads and the number of tasks being created, as inside the function we are creating more threads. Those extra threads could be achieving a higher thread overhead time, because it takes time creating them.

## 8 Printing each thread

Here we can see the result of printing the id of each thread that we have generated.

A terminal window with a dark background and light-colored text. The prompt is 'dvlopez@david:~/Escritorio/PP/P1/lab1\_submit\$'. The command './main' has been executed. The output consists of eight lines, each starting with 'Hey I'm the thread with id' followed by a number: 1, 6, 0, 7, 2, 4, 3, and 5. Below these, the text 'convertGRB2RGBA time for 100 iterations = 147030us' is displayed, followed by 'Executed!! Results OK.' on the final line.

```
dvlopez@david:~/Escritorio/PP/P1/lab1_submit$ ./main
Hey I'm the thread with id 1.
Hey I'm the thread with id 6.
Hey I'm the thread with id 0.
Hey I'm the thread with id 7.
Hey I'm the thread with id 2.
Hey I'm the thread with id 4.
Hey I'm the thread with id 3.
Hey I'm the thread with id 5.
convertGRB2RGBA time for 100 iterations = 147030us
Executed!! Results OK.
```

Figure 8: Image from the terminal showing the different threads that have been created.

We can see that there are 8 threads that have been generated and that will be working concurrently in the *for*. To print the IDs of each thread just once we have to split the *omp parallel for* in two. The first part is adding *omp critical* inside of *omp parallel* and then we will be able to print the threads inside the *omp critical* section. After that we have to add outside of *omp critical* an *omp for* with the *for* loop.