EASYPAP: a Framework for Learning Parallel Programming

Alice Lasserre, Raymond Namyst, Pierre-André Wacrenier

▶ **To cite this version:**

Alice Lasserre, Raymond Namyst, Pierre-André Wacrenier. EASYPAP: a Framework for Learning Parallel Programming. 2021. hal-03126887v1

HAL Id: hal-03126887

https://hal.science/hal-03126887v1

Preprint submitted on 1 Feb 2021 (v1), last revised 9 Aug 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EASYPAP: a Framework for Learning Parallel Programming

Alice Lasserre[a,*], Raymond Namyst[a], Pierre-André Wacrenier[a]

[a]*CS dept., University of Bordeaux, Inria Bordeaux Sud-Ouest, Talence, France*

Abstract

This paper presents EASYPAP, an easy-to-use programming environment designed to help students to learn parallel programming. EASYPAP features a wide range of 2D computation kernels that the students are invited to parallelize using Pthreads, OpenMP, OpenCL or MPI. Execution of kernels can be interactively visualized, and powerful monitoring tools allow students to observe both the scheduling of computations and the assignment of 2D tiles to threads/processes. By focusing on algorithms and data distribution, students can experiment with diverse code variants and tune multiple parameters, resulting in richer problem exploration and faster progress towards efficient solutions. We present selected lab assignments which illustrate how EASYPAP improves the way students explore parallel programming.

Keywords: parallel programming, visualization, monitoring, education, OpenMP, MPI

1. Introduction

During the last decade, the High Performance Computing community had a hard time coping with the evolution of parallel architectures toward massively parallel heterogeneous multicore machines. In fact, all software developers are currently concerned about this manycore trend which has impacted every commodity hardware, from smartphones to desktop machines. To get the most out of nowadays computers, people must be trained to parallel programming.

It is no surprise that integrating HPC into undergraduate and postgraduate courses to expose all students to basic parallel programming skills has been identified as a priority by the *European Technology Platform for HPC* in their 3rd Strategic Research Agenda [11].

Unfortunately, learning parallel programming is intrinsically more difficult than learning sequential programming, especially because students lack convenient and easy-to-use tools to get familiar with nondeterminism and to visualize what happened during a parallel execution.

We present EASYPAP, our attempt to provide students with a simple and attractive programming environment to facilitate their discovery of the main concepts of parallel programming. EASYPAP is a frame-

*Corresponding author

Email addresses: alice.lasserre@etu.u-bordeaux.fr (Alice Lasserre), raymond.namyst@u-bordeaux.fr (Raymond Namyst), pierre-andre.wacrenier@u-bordeaux.fr (Pierre-André Wacrenier)

work providing interactive visualization, real-time monitoring facilities, and off-line trace exploration utilities. Students focus on parallelizing 2D computation kernels using Pthreads, OpenMP, OpenCL, MPI, intrinsics instructions, or a mix of them. EASYPAP was designed to make it easy to implement multiple variants of a given kernel, and to experiment with and understand the influence of many parameters related to the scheduling policy or the data decomposition. During our undergraduate and postgraduate lab sessions, students enjoyed the feedback provided by the graphical tools and were able to gain a deeper understanding of both parallel programming and computer architecture.

The remainder of this paper is organized as follows. We describe the EASYPAP environmment and its associated tool in Section 2. In Section 3, we present how we use EASYPAP to enhance our parallel computing lectures by introducing new parallel concepts in a lively, interactive manner. Examples of OpenMP, OpenCL and MPI assignments are detailled in section 4. We discuss how our pedagogical approch fits in Bloom's taxonomy and present an evaluation of EASYPAP in Section 5. We position our approach with respect to related work is Section 6. Finally, we dress concluding remarks in Section 7.

2. The EASYPAP Framework

EASYPAP is a C programming environment that relies on the SDL library [2] to interactively render the results of 2D computations at run time. It is available on both Linux and Mac OS X systems and can be downloaded from [19]. EASYPAP's main philosophy is to let students focus on computation kernels while hiding most of the implementation details related to program initialization, code instrumentation and interactive display. The `main` program loop is thus controlled by EASYPAP.

2.1. Kernels and variants

In EASYPAP, functions performing computations on images are called *kernels*. EASYPAP comes with a large set of predefined kernels (e.g. *Transpose, Invert, Blur, Pixelize, Game Of Life, Mandelbrot, Abelian Sandpile*). New kernels can easily be added by placing, in the kernels' subdirectory, a `C` file defining at least one function (e.g. `mykernel_compute_variant`) and recompiling EASYPAP.

Let us take the simple `spin` kernel, which colors the pixels of an image according to their polar coordinates, as an illustration. At each iteration, the resulting image is a wheel drawn with a given base angle. This angle is slightly increased between iterations, giving the illusion of a spinning wheel accross multiple iterations.

Fig. 1 shows a sequential implementation of the `spin` kernel. The outer loop (line 3) performs the requested `nb_iter` iterations in a row. In interactive mode, this variable is assigned the value 1 by default, so that the graphical window is refreshed after each iteration. However, this variable may be interactively changed to display one image every `nb_iter` iterations.

```
1 void spin_compute_seq (unsigned nb_iter)
2 {
3   for (int it = 1; it <= nb_iter; it++) {
4     for (int y = 0; y < DIM; y++)
5       for (int x = 0; x < DIM; x++)
6         cur_img (y, x) = compute_color (y, x);
7     rotate (); // change the drawing angle for the next iteration
8   }
9 }
```

Figure 1: Sequential version of kernel `spin`

When running in performance mode (see Section 2.3), no display is involved and the `nb_iter` variable is set to the total number of iterations requested by the user.

Lines 4–6 illustrate how the contents of the image are accessed during an iteration. For the sake of simplicity, EASYPAP works on square ($\texttt{DIM}^2$) shape images. The pixels of the image are accessed through the `cur_img(row, col)` macro. Here is how to run the `seq` variant of the `spin` kernel on a 2048×2048 image:

```
easypap --kernel spin --variant seq --size 2048
```

This action brings a window on the screen which displays an animation consisting of the series of images computed at each iteration. The animation can be paused, or can be slightly accelerated by skipping frames. Under the hood, EASYPAP uses the `dlsym` Unix dynamic linker facility to find and call the appropriate function (i.e. `spin_compute_seq`).

Since the computation of any (i, j) pixel can be performed independently, the `spin` kernel can be trivially parallelized. To develop a straightforward OpenMP version designed as an incremental evolution of the sequential variant, we can simply duplicate the sequential variant, rename it `spin_compute_omp`, insert a single "`#pragma omp parallel for`" clause before the **`for`** loop iterating over lines, recompile EASYPAP, and launch:

```
easypap --kernel spin --variant omp
```

The obtained graphical animation allows the students to visually check if this variant produces the expected output and if it runs faster.

The simplicity with which students are able to implement while maintaining many different variants of given kernel is an essential feature of EASYPAP. Indeed, it makes it very convenient to compare variants against each other and explore their robustness when changing some parameters, as we further discuss in the next sections.

2.2. *Online monitoring*

In order to get more feedback about the parallel execution of a variant, the code needs to be slightly instrumented. To do so, sequential portions of code computing image chunks (called tiles) have to be bracketed by calls to `monitoring_{start/end}_tile`.

```
// Tile inner computation
static void do_tile (int x, int y,
                     int width, int height, int thr)
{
  monitoring_start_tile (thr);
  for (int i = y; i < y + height; i++)
    for (int j = x; j < x + width; j++)
      cur_img (i, j) = compute_color (i, j);
  monitoring_end_tile (x, y, width, height, thr);
}

void spin_compute_omp_tiled (unsigned nb_iter)
{
#pragma omp parallel
  for (int it = 1; it <= nb_iter; it++) {
#pragma omp for collapse(2) schedule(static)
    for (int y = 0; y < DIM; y += TILE_SIZE)
      for (int x = 0; x < DIM; x += TILE_SIZE)
        do_tile (x, y, TILE_SIZE, TILE_SIZE,
                 omp_get_thread_num ());
#pragma omp single
    zoom ();
  }
}
```

Figure 2: Typical example of instrumented code using calls to `monitoring_start_tile` and `monitoring_end_tile`

Fig. 2 shows a typical OpenMP tiled implementation of the `spin` kernel where the `do_tile` function has been instrumented. This function sequentially computes all the pixels inside an arbitrary rectangle defined by `(x,y,width,height)`. Note that our `omp_tiled` variant uses only square tiles, in the general case any rectangle shape is relevant. The last parameter is the rank (from 0 to $\#threads - 1$) of the thread which computes the tile. With OpenMP, we just pass `omp_get_thread_num()`. To ease the use of tiling EASYPAP provides two global variables `NUM_TILES` and `TILE_SIZE` which may be set using command line options. EASYPAP will deduce the value of the unspecified variable thanks to the equation `DIM` = `NUM_TILES` × `TILE_SIZE`.

Once the code has been instrumented, real-time monitoring can simply be activated:

```
easypap --kernel spin --variant omp_tiled --tile-size 64 --monitoring
```

The monitoring mode pops up two additional side windows as displayed in Fig. 3.

The **Activity Monitor window** reports the real-time load of each CPU. This load is a percentage representing the amount of time spent in computations over the duration of the iteration. In constrat with

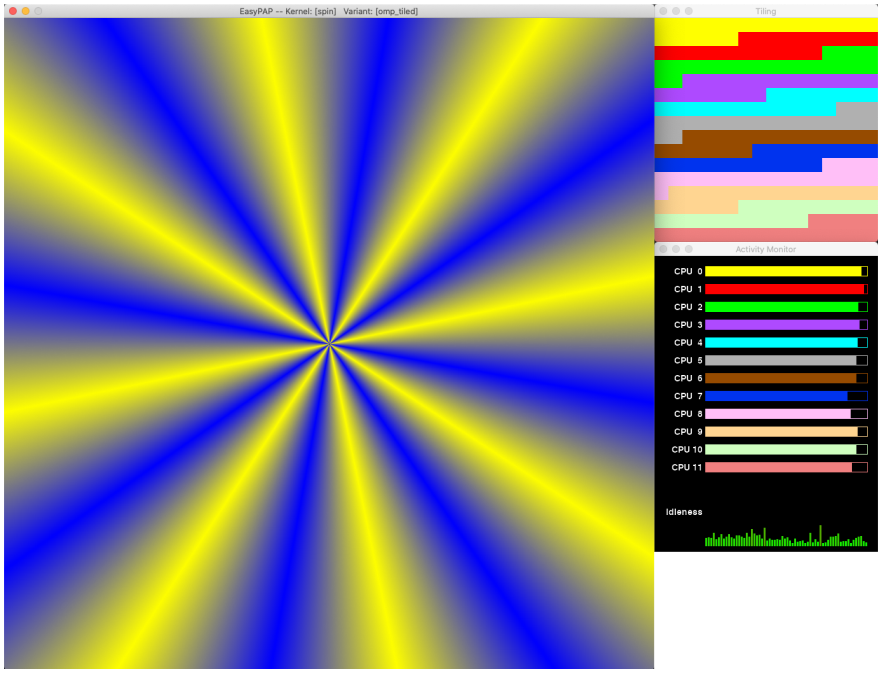


Figure 3: In addition to the main graphical kernel output, the monitoring mode introduces two additionnal windows: a tiling window (top) and a CPU monitoring window.

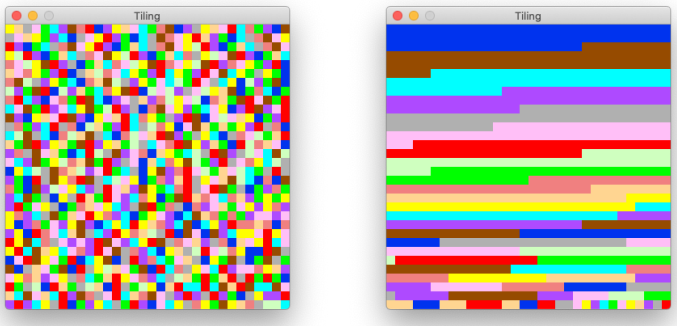
system wide perfmeters, the activity monitor only reflects the kernel behavior. For instance, the overhead of updating the main graphical window is not taken into account. At the bottom of the window, a history diagram reports the evolution of *cumulated idleness* over time.

The **Tiling window** reflects the way tiles have been assigned to CPUs at each iteration. Each CPU is assigned the same color as in the *Activity Monitor* window. By observing Fig. 3, we see that the tiles have been assigned to threads in contiguous chunks, in accordance to the *static* loop scheduling policy.

The tiling window is a precious tool to graphically observe and compare the different loop scheduling policies of OpenMP. In Fig. 4, we examine two loop scheduling policies through the tiling window. Fig. 4a reveals the opportunistic nature of the `schedule(dynamic)` clause, whereas Fig. 4b shows how the size of chunks assigned to threads decreases over time with the `schedule(guided)` policy.

2.3. Performance mode

In order to accurately benchmark and compare the performance of multiple variants, we need to completely eliminate the overhead of graphical updates. When invoked with the `--no-display` option, EASYPAP runs silently and reports the overall wall clock time after completion of the requested number of iterations.



(a) policy = `schedule(dynamic)` (b) policy = `schedule(guided)`

Figure 4: During execution, students observe how the OpenMP loop scheduling policy impacts the assignment of tiles to threads.

```
easypap --kernel spin --variant omp_tiled \
        --tile-size 64 --iterations 10    \
        --no-display
10 iterations completed in 66.282ms
```

Moreover, the completion time, together with all execution and configuration parameters, are reported in a *Comma Separated Values* (`CSV`) file. Students can customize simple python scripts to automate their experiments by specifying parameter ranges, as illustrated in Fig 5. Then students can produce the desired graph or histogram from the data thanks to the python script EASYPLOT (based on the data visualization libraries `Seaborn / Matplotlib` [23, 16]). Several options of EASYPLOT are to select data from the performance file, where possible these options have similar names to the options in `easypap`. Other options are available to specify figure-level or axes-level parameters. For instance Fig. 6 was produced thanks to the following command :

```
easyplot.py --kernel spin --variant omp_tiled omp_tiled_vec # selects rows \
            -x threads  # sets the x-axis in order to get a scalling plot  \
            -y speedup  # plots speedup ratios against...                  \
            --refTimeVariant seq #                    ... the seq variant \
            --col dim --row variant # produces an array of subplots
```

A key feature of EASYPLOT is that speedup ratios and legends are automatically generated from the data. Once data have been filtered, constant parameters are put aside, and the names of plotlines are set using the remaining ones (see Fig. 6). This guarantees that experiments conducted in different conditions will not silently be incorporated in the same graph.

2.4. Post mortem trace analysis

Although the monitoring facilities greatly help to detect and understand flaws in the execution of kernels, it cannot always capture some subtle properties such as the heterogeneity of tasks duration or the correct implementation of task dependencies. When a deeper analysis is required, students use the `--trace`

```
from expTools import *

easypap_options = {
    "--kernel ": ["spin"],
    "--variant ": ["omp_tiled_vec", "omp_tiled"],
    "--iterations ": [10],
    "--size ": [512, 1024],
    "--tile-size ": [64]
}

omp_env = {
    "OMP_NUM_THREADS=": [1] + list(range(2, 13, 2)),
    "OMP_SCHEDULE=": ["static", "dynamic", "guided"]
}

execute('easypap', omp_env, easypap_options, num_runs=30)

# Run sequential variant in order to compute speedup ratios
options["--variant "] = ["seq"]
omp_env = {"OMP_NUM_THREADS=": [1]}
execute('easypap', omp_env, easypap_options, num_runs=3)
```

Figure 5: Typical experiments automation script

option to record tile-related profiling events at execution time (i.e. start/end time, tile coordinates, cpu) into a trace file:

```
easypap --kernel spin --variant omp_tiled --traces --no-display --iterations 2
```

To visually explore and interact with the collected trace, we provide the EASYVIEW utility (Fig. 7). Its graphical interface is subdivided in two parts.

The left side presents a view widely adopted by many trace viewers: a Gantt chart displays per-CPU sequences of tasks for a selectable range of iterations. Tiles computed by the same CPU have the same color, and are displayed on the same timeline. When moving the mouse over a task, a pop-up bubble displays the task duration.

The right side displays a reduced view of the surface computed at the selected iteration (see thumbnails of the spinning wheel appearing in Fig. 7). Whenever the x-axis of the mouse intersects tasks in the Gantt chart, the corresponding tiles are highlighted over this reduced image, helping to localize computations. As a consequence, starting on the left side of the Gantt chart and moving smoothly the mouse towards the right side reveals the order in which tiles have been computed.

In addition, students can toggle between this vertical mouse mode and a horizontal mode in which the y-axis of the mouse allows to select a particular CPU and highlights the tiles computed during the

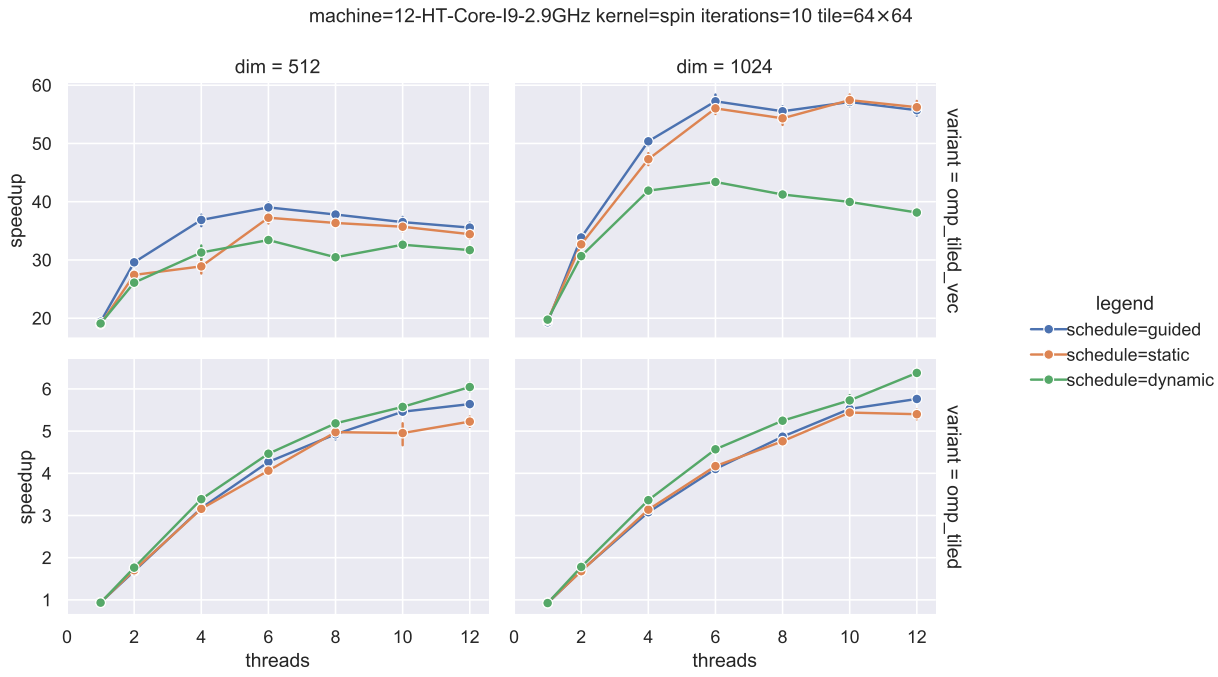


Figure 6: Strong Scaling plots of vectorized and scalar variants of the kernel spin. This graph is generated by EASYPLOT from the data produced by the execution of the script presented in Fig. 5. The common parameters are listed above the graph. We can see that scalar variants scale well, indeed they are compute bounded. Nevertheless, vectorized variants outperform scalar variants although they do not scale very well on small size cases.

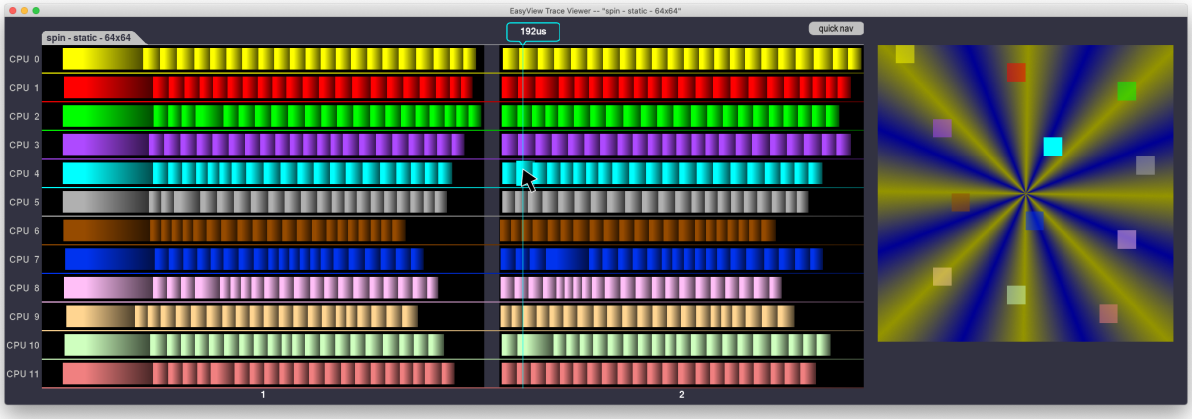


Figure 7: EASYVIEW brings interactive exploration of traces. Moving the mouse over a task in the Gantt diagram displays its duration (bubble at top of window). Tasks intersecting the mouse x-axis have their corresponding tile highlighted over the image thumbnail, allowing to link computations and their data. This trace shows the first two iterations of the OpenMP tiled variant of the `spin` kernel. We can observe that most tasks have a homogeneous duration (around 200μs) at the notable exception of the first tasks computed by each CPU (leftmost part of Gantt chart) which do not benefit from the "*warm cache*" effect.

displayed period. Basically, this allows to observe the "coverage map" of a given CPU during one or multiple iterations, and to check the locality of computations across iterations.

EASYVIEW is a powerful mean for students to understand how the scheduling of computations are performed, to see which image areas are the most time-consuming, to check if the computations were evenly balanced over computing units, and even to track down synchronization issues. We highlight a series of such situations in Section 4.

2.5. GPU Programming with OpenCL

Building on a tight integration with the OpenCL [21] framework, EASYPAP consistently extends the facilities presented in the previous sections to the scope of GPU programming: students can focus their effort on the implementation of OpenCL kernels, interactively watch their execution on screen, record performance numbers in performance mode, or generate execution traces.

Most of the OpenCL initialization code is hidden, saving students from the tedious task of coping with the discovery of hardware platforms, the creation of contexts, buffers and queues, and the compilation of kernels. Moreover, the implementation of EASYPAP exploits data sharing facilities between OpenCL and the Open Graphics Library (OpenGL) to efficiently refresh on-screen display with no data movement. EASYPAP currently supports running OpenCL kernels on a single OpenCL device, which can be selected using environment variables in case multiple devices are available. By default, EASYPAP selects the GPU used by the OpenGL driver, as shown below:

```
[my-machine] easypap --show-ocl
1 OpenCL platforms detected
Platform 0: Apple (Apple)
--- Device 0 : CPU [Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz]
--- Device 1 : GPU [Intel(R) UHD Graphics 630]
+++ Device 2 : GPU [AMD Radeon Pro 560X Compute Engine]
Note: OpenGL renderer uses [AMD Radeon Pro 560X OpenGL Engine]
```

As for CPU kernels, a buffer named `cur_buffer` is pre-allocated on the device and the initial image (if any) is tranferred to this buffer before the computation loop starts. The multiple variants of a kernel (e.g. `spin`) have to be implemented in the same file (e.g. `kernel/ocl/spin.cl`). Figure 8 shows the contents of the `spin.cl` file which features the definition of a default variant of the `spin` kernel. The kernel takes two arguments: the address of the image in GPU's memory, and the value of the drawing angle. This straightforward kernel variant assumes that the caller will request the creation of `DIM` × `DIM` workitems, each workitem being responsible for computing only one pixel (Fig. 8, lines 9–11).

On the CPU side, the students are provided with a "template" C function which serves as the kernel invocation routine. They need to customize this function to pass the appropriate parameters to the OpenCL kernel. In the case of the `spin` kernel, students end up with the function depicted in Figure 9.

```
static unsigned compute_color (int y, int x, const float drawing_angle)
{
  ... // basically a copy/paste of the sequential code
}

__kernel void spin_ocl (__global unsigned *out,
                        const float drawing_angle)
{
  int x = get_global_id (0), y = get_global_id (1);

  out [y * DIM + x] = compute_color (y, x, drawing_angle);
}
```

Figure 8: OpenCL implementation of the `spin` kernel

Note that the code presented in Figures 8 and 9 faithfully represents all students need to write to obtain an effective `spin` OpenCL implementation. Most importantly, OpenCL kernels variants are launched the same way[1] as any CPU variant, with most parameters applying to both targets. This is particularly useful when using scripts to automate experiments. More importantly, it makes students feel confortable when going from OpenMP to OpenCL programs: they are instantly able to run interactive experiments, plot curves, observe monitoring data or inspect executions traces.

The following command line launches the spin kernel with an interactive visualization window:

```
easypap --ocl --kernel spin
```

Whereas the following command line runs 1000 iterations in *perfomance mode* and reports the overall computation time:

```
easypap --ocl --kernel spin --no-display --iterations 1000
```

Like sequential or OpenMP kernels, the execution of OpenCL kernels can be instrumented for monitoring or trace recording purposes. The highlighted lines of code in Figure 9 illustrate how the timing events related to each kernel execution can be easily captured. Note that it is a coarse grain instrumentation, since we have no detail about what happens inside the device (e.g. at the workgroup level). Yet it helps to observe the activity periods of an OpenCL device, especially when used in parallel with CPUs. When implementing more complex kernels, such as hybrid CPU-GPU 2D stencil simulations where ghost regions have to be exchanged at each iteration, each data transfer can be instrumented as well and will appear in the execution trace. This feature will be further explored in Section4.2.

[1]The only minor difference is that the `--ocl` flag must be passed to activate OpenCL-specific initializations.

```
1  unsigned spin_invoke_ocl (unsigned nb_iter)
2  {
3    size_t global[2] = { DIM, DIM };      // We spawn one workitem per pixel
4    size_t local[2]  = { TILEX, TILEY }; // workgroup size
5    cl_event kernel_event;
6
7    for (unsigned it = 1; it <= nb_iter; it++) {
8      clSetKernelArg (compute_kernel, 0, sizeof (cl_mem), &cur_buffer);
9      clSetKernelArg (compute_kernel, 1, sizeof (float), &drawing_angle);
10
11     clEnqueueNDRangeKernel (default_queue, compute_kernel, 2, NULL, global, local,
12                             0, NULL, &kernel_event);
13     clFinish (default_queue);
14     monitor_kernel (kernel_event, 0 /* x */, 0 /* y */,
15                     global[0] /* width */, global[1] /*height */, KERNEL_OP);
16     clReleaseEvent (kernel_event);
17
18     rotate ();  // change the drawing angle for the next iteration
19   }
20   return 0;
21 }
```

Figure 9: OpenCL Launcher function responsible for invoking the `spin` kernel. Highlighted lines are optinal and llustrate how OpenCL code can easily be instrumented for monitoring purposes.

2.6. *Distributed Programming with MPI*

EASYPAP was designed right from the start to support distributed programming using the Message Passing Interface (MPI). The `easypap` script leverages the standard `mpirun` process launcher to spawn multiple EASYPAP processes through a `--mpirun` configuration flag. As an illustration, here is how to run a set of four MPI-enabled EASYPAP processes:

```
easypap --kernel spin --variant mpi --mpirun "-np 4"
```

Once initialized, all MPI processes execute the `spin_compute_mpi` function in a SPMD[2] manner and can use any MPI routine to discover their rank, exchange data, etc. The `cur_img` buffer is allocated inside each process, and if initial pixel values are loaded from the disk, it gets replicated in every process before the actual computation starts.

Figure 10 illustrates how a MPI-variant of the `spin` can be implemented. Since the computation load is the same for every pixel of the image, we use a block-decomposition method where each process will be in charge of a distinct horizontal stripe of the image. This is done in the `spin_init_mpi` initialization function (line 5–15), where the bounds of the stripe assigned to the current process are determined. The actual computation of the stripe inside each process is parallelized using an OpenMP tile-based work distribution (line 22). Eventually, after a series of `nb_iter` iterations, the contributions of all processes are brought back

[2]Simple Program Multiple Data

on the master process using a `MPI_Gather` collective operation (lines 30–31).

```
1  static int mpi_y    = -1; // first line to be computed
2  static int mpi_h    = -1; // height of stripe to be computed
3  static int mpi_rank = -1, mpi_size = -1;
4
5  void spin_init_mpi (void)
6  {
7    easypap_check_mpi (); // make sure --mpirun option was specified on command line
8
9    MPI_Comm_rank (MPI_COMM_WORLD, &mpi_rank);
10   MPI_Comm_size (MPI_COMM_WORLD, &mpi_size);
11
12   mpi_y = mpi_rank * (DIM / mpi_size);
13   mpi_h = (DIM / mpi_size);
14   // current process must process lines in interval [mpi_y..mpi_y+mpi_h-1]
15 }
16
17 unsigned spin_compute_mpi (unsigned nb_iter)
18 {
19   for (unsigned it = 1; it <= nb_iter; it++) {
20     // We could simply call do_tile (0, mpi_y, DIM, mpi_h, 0)
21     // to perform our computation sequentially...
22 #pragma omp parallel for collapse(2) schedule(runtime)
23     for (int y = mpi_y; y < mpi_y + mpi_h; y += TILE_SIZE)
24       for (int x = 0; x < DIM; x += TILE_SIZE)
25         do_tile (x, y, TILE_SIZE, TILE_SIZE, omp_get_thread_num ());
26
27     rotate ();
28   }
29
30   MPI_Gather ((mpi_rank == 0 ? MPI_IN_PLACE : image + mpi_y * DIM), mpi_h * DIM,
31               MPI_INT, image, mpi_h * DIM, MPI_INT, 0, MPI_COMM_WORLD);
32
33   return 0;
34 }
```

Figure 10: Hybrid MPI + OpenMP implementation of the `spin` kernel. In the `spin_init_mpi` initialization function, which is automatically invoked before the actual computation starts, each process determines which region (horizontal stripe) it is responsible for. Since the kernel is trivially parallel, iterations are computed independently by each process, and data transfers are performed only when all the pixels need to be gathered on the master process (lines 30–31).

When running EASYPAP in interactive mode, only the graphical window of the master MPI process is displayed by default. However, for debugging purposes, it is possible to activate the display of every process' window. Moreover, the monitoring windows (tiling window and activity monitor) of each process can be displayed as well, using the `--debug` flag:

```
OMP_NUM_THREADS=6 easypap --kernel spin --variant mpi   \
                          --mpirun "-np 2" --monitoring \
                          --debug M
```

Figure 11 displays a screenshot captured during the execution of our MPI+OpenMP variant of `spin` using two MPI processes. The three windows on the right (resp. on the left) belong to process of rank 0

(resp. rank 1). Inside each process, 6 OpenMP threads were used to process tiles in parallel. By inspecting the tiling window of each process, we can clearly see that process 0 work on the upper half the image and process 1 on the bottom half. However, by looking at the main window of each process, we observe that only the master process has the complete image. This is because `MPI_Gather` is called after each iteration and is followed by a screen refresh in interactive mode. In contrast, process of rank 1 has only the pixels it has computed.

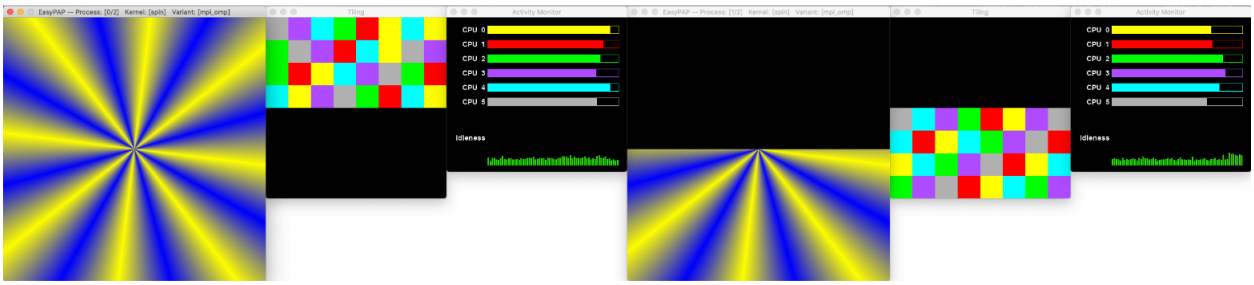


Figure 11: Snapshot of a MPI+OpenMP variant of the `spin` kernel. The execution uses two MPI processes, each hosting 6 OpenMP threads.

Trace generation facilities are also available for MPI programs: timing events are recorded in a separate file for each process. These traces can then be vizualized with EASYVIEW, either individually or using a side-by-side comparison to diagnose load balancing issues for instance.

3. Transmitting knowledge about HPC with EASYPAP

Although EASYPAP was primarily designed as a practice framework to help student to implement and experiment with parallel computations, it proved to be also a great teachers' companion during *parallel programming* lectures, in a pure "transmissive" teaching mode.

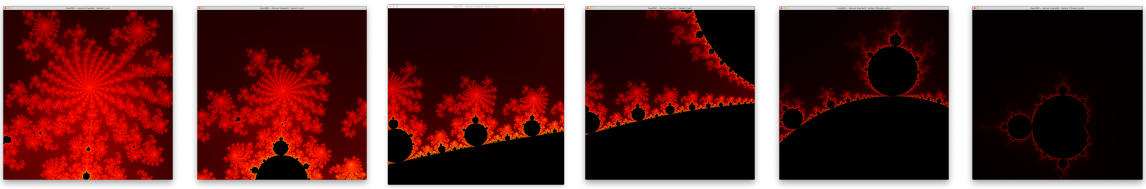
We claim that a lot of parallel concepts can be illustrated using a well-chosen, visually attractive kernel executed under monitoring mode. Observing interactively the impact of changing some parameters (e.g. work distribution strategy, tile size, minor code modification), facilitates students understanding by involving their long term visual memory.

EASYPAP can be used during lectures to motivate graduate students by showing a teaser of what they will achieve during lab sessions, or to demystify advanced language constructs in a graphical manner (e.g. behavior of the new OpenMP 5 *nonmonotonic* scheduling policy). Moreover, it can also be used to introduce parallelism to undergraduate (and younger) students who have never heard about parallelism before, as discussed in the next section.

3.1. Popularizing the basic concepts of parallelism

The computation of the Mandelbrot set [9] is a good example of a simple 2D kernel that can visually illustrate the notion of workload. The sequential implementation of our `mandel` kernel is very similar to the code of `spin` depicted in Figure 1, except that the color of each pixel is determined by the number of terms of a series that were computed before either observing divergence or crossing an arbitrarily chosen threshold. The second difference is that, after each iteration, we slightly change the coordinates of the complex plane area displayed on screen to obtain a zoom effect between frames. We actually use a "zoom out" strategy to start from a close-up view a fractal curve and to progressively unveil the whole mandelbrot set, as shown in Figure 12.

When running the sequential variant of `mandel` in interactive mode, the animation plays quite smoothly during the very first iterations, but a noticeable slowdown takes place as soon as a significant portion of the Mandelbrot set enters the frame (see the black area at the bottom of snapshot in Subfigure 12b). This slowdown goes worse as the Mandelbrot set keeps occupying a bigger portion of the frame (from iteration 30 to approximately 200 in Figure 12). Note that even if students are not familiar with Mandelbot fractals, as soon as they are told that black pixels (i.e. belonging to the Mandelbrot set) require much more computations than other pixels, they easily understand where the slowdown comes from.



(a) 1st iteration (b) 30th iteration (c) 75th iteration (d) 120th iteration (e) 200th iteration (f) 300th iteration

Figure 12: Evolution of the Mandelbrot displayed area across iterations.

In a second step, we introduce the benefits of parallelism by running a simple, parallel version where the frame is evenly divided into stripes, one stripe being assigned to a distinct CPU (Figure 13). At this point, showing the (OpenMP) source code is not needed. We only want students to feel the acceleration in comparison with the previous sequential run. In order to draw attention on the way the workload was assigned to CPUs, we run EASYPAP under monitoring mode (Figure 13). The execution is now noticeably faster, achieving an average frame rate of 26 fps (*frames per second*) versus 9 fps for the sequential version.

However, the activity monitor panel (Figure 13) clearly reveals a strong load imbalance between CPUs. The static distribution of work is indeed inappropriate because the large black area at the bottom of the image, which contains a lot of pixels belonging to the Mandelbrot set, involves much more computations than other areas.

At this point, an interesting discussion can take place with the audience about what would be a "good"

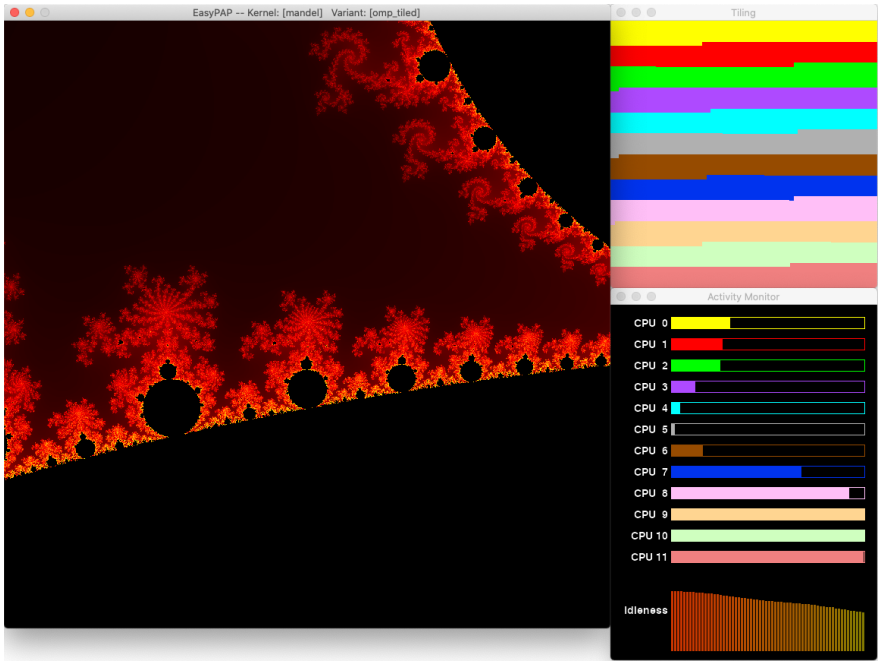


Figure 13: Screenshot (at iteration 100) of the OpenMP variant of the `mandel` kernel with a static distribution of tiles.

work distribution strategy. Even if students have no specific knowledge about the various OpenMP scheduling policies, the discussion quickly engages toward fairer ways of assigning small tiles of the frame, such as cyclic or dynamic distributions.

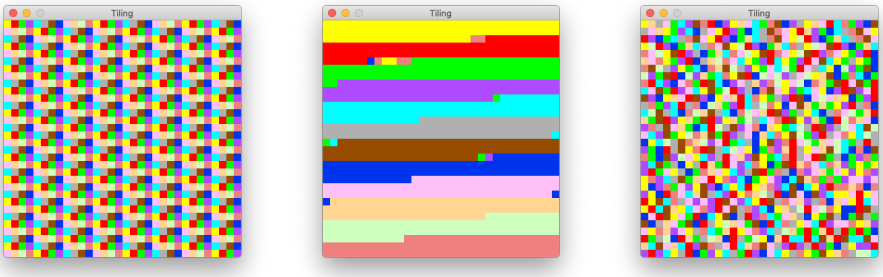
3.2. Understanding work distribution policies

As discussed in Section 2.2, the tiling window is particularly useful for observing and comparing different work distribution policies. Similarly to the `spin` kernel (Section 2.2), our OpenMP variant of `mandel` uses a tile-based decomposition, the assignment of tiles being customizable at run time:

```
#pragma omp for collapse (2) schedule (runtime)
  for (int y = 0; y < DIM; y += TILE_SIZE)
    for (int x = 0; x < DIM; x += TILE_SIZE)
      do_tile (x, y, TILE_SIZE, TILE_SIZE, omp_get_thread_num ());
```

Changing the loop scheduling policy is thus a matter of setting the `OMP_SCHEDULE` environment variable to the desired policy. Figure 14 shows the output of the tiling window when using three different loop scheduling policies.

On Subfigure 14a, we observe a cyclic distribution of tiles among threads resulting from the use of the `static,1` scheduling directive. When tile size is kept sufficiently small, this strategy brings a neat performance improvement over the block-static distribution because "heavyweight" tiles are somewhat homogeneously distributed among CPUs.



(a) static,1 (b) nonmonotonic:dynamic (c) dynamic

Figure 14: The Tiling Window allows to compare the different OpenMP loop scheduling policies on the actual distribution of tiles.

The `mandel` kernel offers a good opportunity to study the behavior of the "nonmonotonic:dynamic" scheduling directive recently introduced in OpenMP 5). Under this strategy, tiles are initially assigned to threads in a static manner. But as soon as some threads go idle, they steal work to overloaded threads. This is clearly visible in Subfigure 14b, where the distribution looks like a static one except that we can distinguish a few stolen tiles located at the ending bounds of static chunks.

Switching to a purely dynamic policy (Subfigure 14c) further enhances performance, as distributing the tiles in a gready manner proved to be the best strategy for the `mandel` kernel. After about a hundred of iterations, students observe interesting patterns appearing in the *tiling window*, as spotted in Fig. 15.

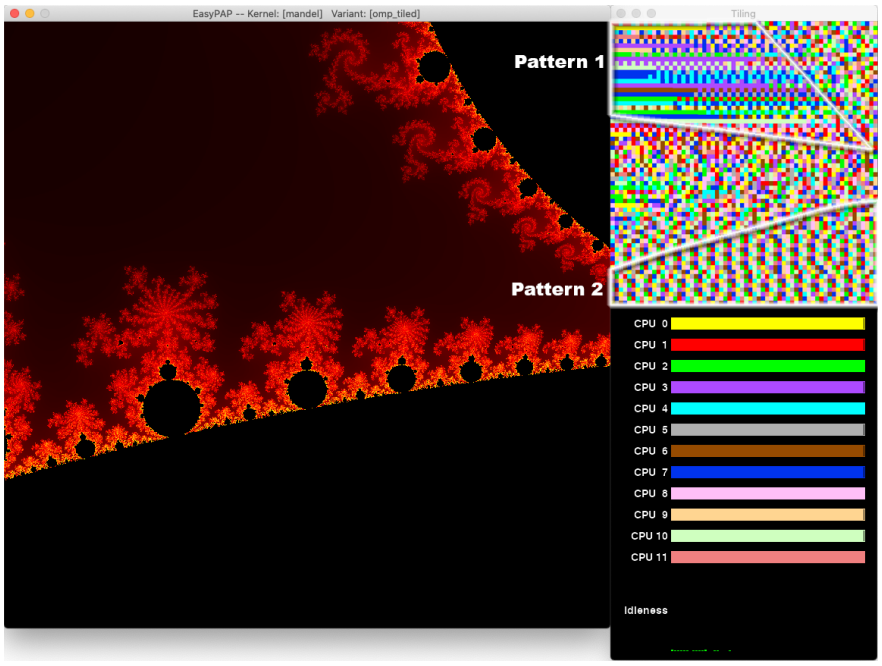


Figure 15: When using OpenMP dynamic loop scheduling of small tiles, the tiling window reveals two noticeable patterns.

Pattern 1 reveals horizontal stripes of the same color together with a few stripes featuring an alternation of two colors. These stripes correspond to one or two threads computing several tiles in a row. Such a

situation happens because 1) these tiles correspond to areas located far away from the Mandelbrot set, where computations take only a few iterations to complete and 2) the other threads are all busy computing time-consuming tiles in the top-right black corner.

In contrast, **Pattern** 2 features a quasi-perfect cyclic distribution of colors. This is due to the fact that all tiles require the same amount of (heavy) computations. Therefore, the dynamic distribution turns into a regular, cyclic one in such areas.

3.3. Investigating performance issues

The previous section illustrates how real-time monitoring tools help students to better understand a large variety of parallel concepts. In situations where more information about temporality is needed, analyzing execution traces interactively can typically bring useful insights about the causes of an underperforming kernel.

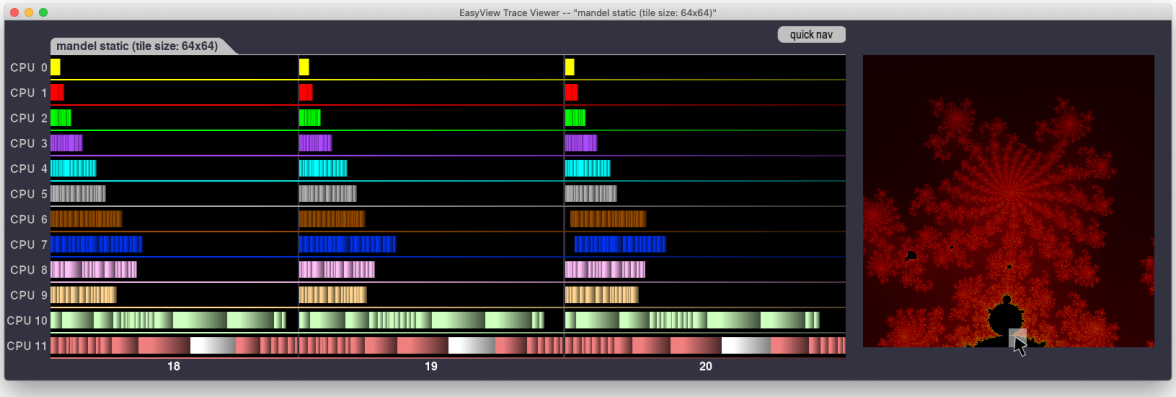


Figure 16: Moving the mouse on the rightmost thumbnail reveals the tasks (highlighted in white color) that have worked on the tile under the mouse pointer.

Let us take our first parallel variant of `mandel` (static distribution of tiles) as an example. By executing the application with the `--trace` flag, we obtain a trace file that we can analyze using EASYVIEW. Figure 16 shows the scheduling of tasks during three successive iterations. Students can easily observe three phenomena. First, the Gantt chart exhibits a tremendous variability of tasks' durations. The longest tasks (approx. 4 680µs) are 130 times longer than the shortest (approx. 35µs) ones. Second, by moving the mouse over the image thumbnail, the tasks responsible for the associated tile are highlighted, which allows to check what tiles of the image were the most demanding, for instance. In the situation spotted in Figure 16, we observe that the tile pointed by the mouse cursor correspond to long-lasting tasks that were executed by CPU 11. Finally, the Gantt chart unambiguously reveals what causes such a work imbalance:

time-consuming tasks are almost exclusively executed on CPUs 10 and 11, which demonstrates that the block-static distribution of tiles is definitely inappropriate in this case.

As discussed in previous Subsection, performance increases when switching from the block-static distribution of tiles to a cyclic one. Switching to the *nonmonotonic:dynamic* loop scheduling policy further enhances performance, but the reason behind this improvement is not obvious. This is where the "trace comparison" capability of EASYVIEW is useful: the task Gantt chart of both scheduling strategies can be aligned iteration per iteration and compared.

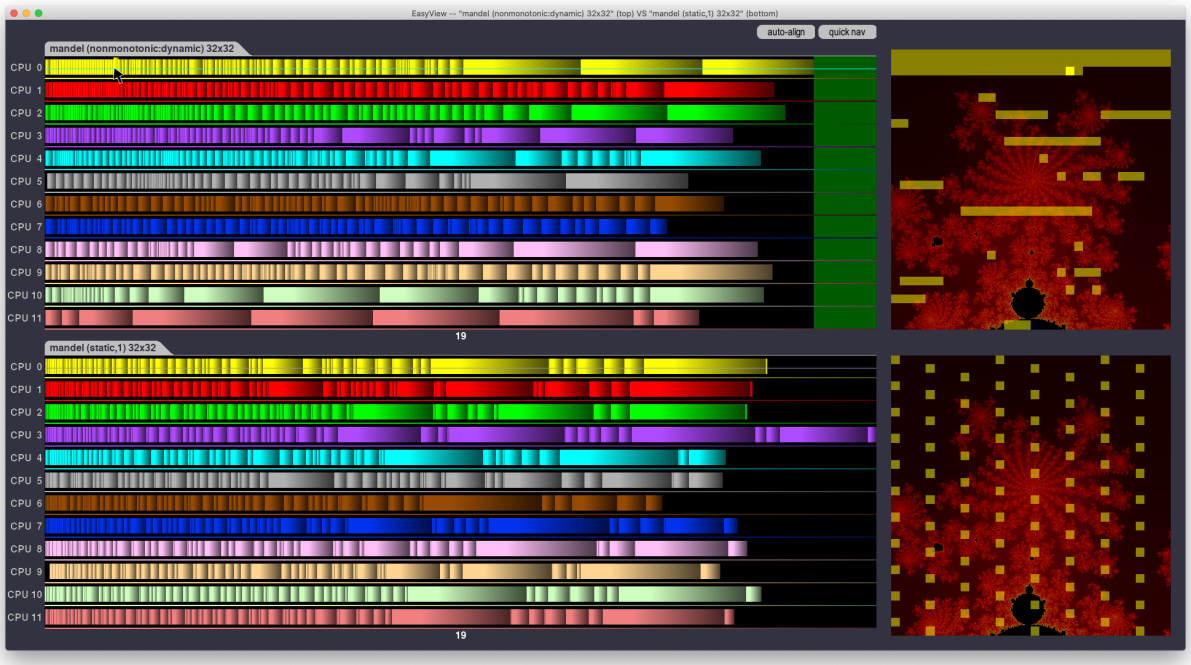


Figure 17: Two executions traces are compared using EASYVIEW. The traces are "re-aligned" so that each iteration virtually begins at the same time on both sides. The green zone indicates the amount of time saved by the fastest execution.

Figure 17 displays the sequences of tasks executed at iteration 19 of the `mandel` kernel using respectively a cyclic distribution (bottom trace) and a nonmonotonic:dynamic distribution (top trace). The "*coverage mode*" feature of EASYVIEW (see Section 2.4) allows us to display the set of tiles processed by CPU 0 during this iteration. One can observe that the dynamic strategy does a better job (although not perfect) in sharing the computation load among CPUs: the cyclic distribution can lead to situations where an "unfortunate" CPU has more work than its peers (CPU 4 in this case).

A second interesting observation comes from the thumbnails on the right side: the students immediately recognize the cyclic assignement of tiles to CPU 0 in the bottom thumbnail, while the top thumbnail reveals that CPU 0 has stolen a lot of tiles after completing its initial bunch of tiles under the dynamic policy.

3.4. Encouraging a scientific experimental approach

The behavior of a program on a parallel machine is difficult to predict, even for a specialist. Indeed, parallel machines are now complex systems that deserve to be studied because high-performance is required. Our pedagogy is based on program optimization in order to understand the behavior of a parallel computer: we teach students to get the quintessence of computers by making them code different implementations of the same kernel. However, observing the behavior of an implementation is not trivial because it can be sensitive to many parameters (e.g. tile shape, scheduling strategy, initial configuration). Also we have to teach students to do experiments to compare implementations or understand the role of a parameter just as we do as researchers. For this purpose we have written an experiment guide that describes how to explore the parameter space using scripts and plots. In this guide we illustrate that the obtained graphs can be used for parameter optimization, implementation comparison as well as for scientific presentation.

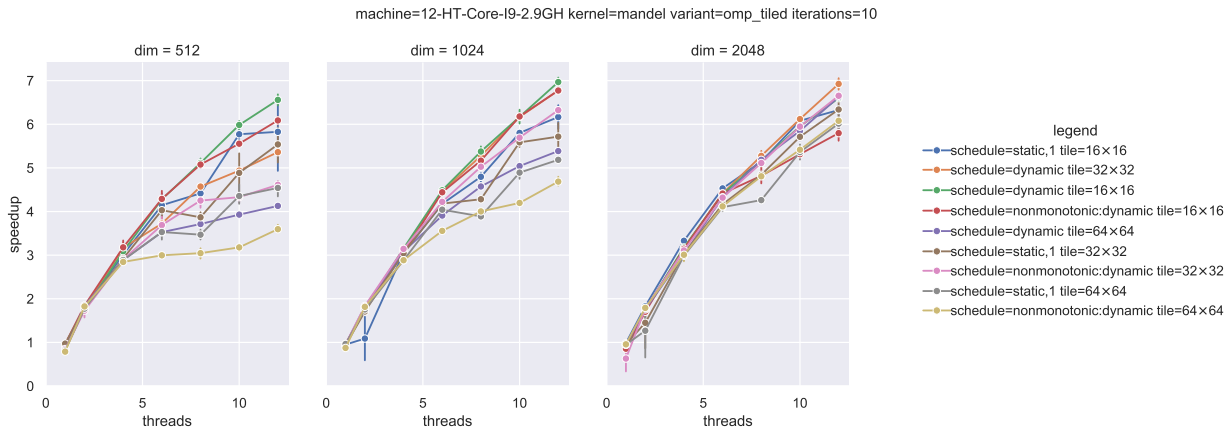
In the tutorial we present how to compare different scheduling policies with the variant `omp_tiled` of `mandel` kernel . For this purpose we give the scripts to produce the data needed to explore different parameters (tile size, scheduling policy, image size) and EASYPLOT's otions to produce a graph such as Fig. 18a. Then we give advices to students on how to present graphs in a report. Indeed, the graph Fig. 18a has too many plotlines to report clearly the situation: it must be simplified by eliminating redundant information or by zooming on few details. We also advise students to formulate hypotheses and to build a scientific description of the graph based on evidences like traces or facts like the behavior of computer hardware. In the tutorial we give some concrete examples like the following one based on figures 18b and 18c which allow to highlight interesting phenomena.

In figure 18b we see that *nonmonotonic:dynamic* and *static,1* policies have identical performances up to 6 threads but that beyond that the nonmonotonic:dynamic policy takes the upper hand. A hypothesis may be that the static policy behaves badly with hyperthreading (indeed the computer consists of 6 physical cores with 2 logical processors per core) - however thanks to the traces presented in figure 17 we know that the difference is due to load balancing.

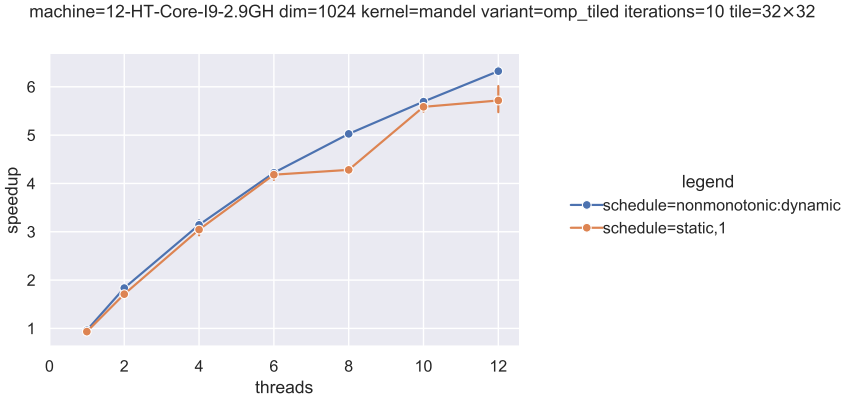
Another detail deserves to be analyzed : on the figure 18b we see that the policy dynamic performs better than the nonmonotonic:dynamic one. Let us look carefully at the trace of the nonmonotonic:dynamic policy depicted figure 17 : we observe that cpu 5 and 7 remain inactive while a task seems to be available. This task will be executed late by CPU 1 whereas with a truly dynamic policy it would have been executed as soon as possible. Once again the difference in performance is explained by a difference in load balancing.

4. Example of assignments

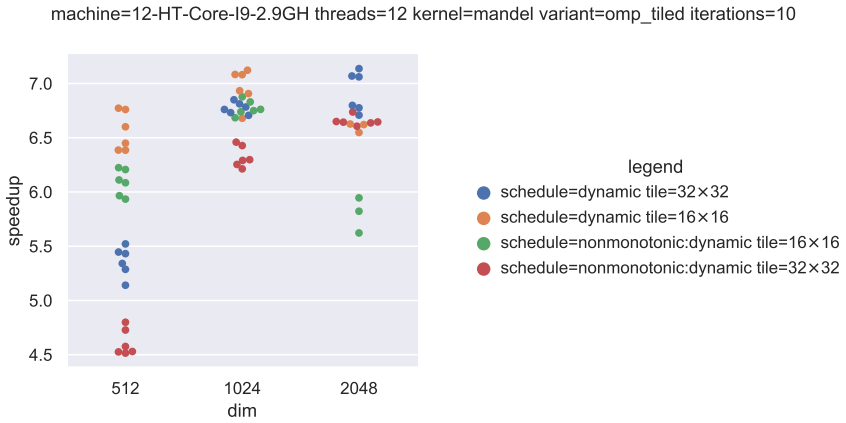
We have used EASYPAP and EASYVIEW both with undergraduate students during parallel programming introduction courses, and with postgraduate students during parallel and distributed computing



(a) Parameters exploration through multiple plots.



(b) Zoom in on `static,1` versus `nonmonotonic:dynamic`.



(c) Zoom in on `dynamic` versus `nonmonotonic:dynamic`.

Figure 18: Some plots given in the experiment tutorial.

courses focussed on more advanced features of multicore, GPU and cluster programming. Even if students usually start with simple, straightforward implementations of basic kernels, they quickly dive into more subtle codes where they encounter bugs and performance issues. Using various case studies, we now explore to what extent EASYPAP and EASYVIEW help students to better understand the behavior of their code and visualize things which are traditionally difficult to observe.

4.1. Multicore programming assignments

After a first hands-on session during which undergraduate students discover the EASYPAP environment using simple kernels, including `spin` and `mandel`, their next assignments are devoted to implementing more complex OpenMP kernels (Game of Life, Abelian Sandpile, Image Blur, etc.) which involve more advanced synchronization schemes and/or additionnal data structures. We detail two of these assignements in the remaining of this section.

4.1.1. Picture Blurring: a simple 2D stencil code

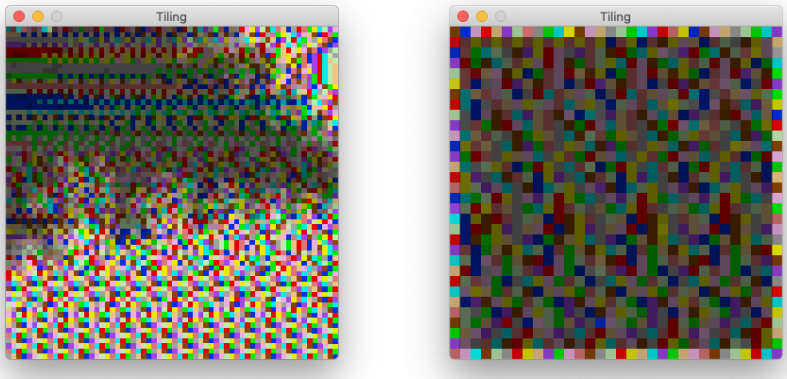
During their discovery of parallel computing, our students are quickly exposed to simulations involving *Stencil* computations. We use an assignment based on a *Picture Blurring* kernel to introduce students to the parallelization of 2D stencil codes. The sequential version of the `blur` kernel (Fig. 19) uses two images. At each iteration, all pixels from the 3×3 square centered in (i, j) are read from the first image, and the average is written to the second one. The two images are swapped between iterations (Fig. 19, line 9).

```
unsigned blur_compute_seq (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    for (int y = 0; y < DIM; y++)
      for (int x = 0; x < DIM; x++)
        next_img (y, x) = average_surrounding_pixels (y, x);

    swap_images (); // swap cur_img and next_img
  }
  return 0;
}
```

Figure 19: Sequential version of the `blur` kernel

Since every pixel is read multiple times at each iteration, students are encouraged to implement a tiled parallel version to maximize cache reuse. To avoid out-of-bounds image accesses for pixels located on the borders (which have less than 9 neighbours), their code includes several conditional branches which leads to poor performance.

By observing that tests are only required for tiles located on the edges (i.e. outer tiles), students implement different codes for outer and inner tiles. After implementing this optimization, they can quickly check



(a) mandel (b) blur

Figure 20: In "heat map" mode, the brightness of tiles displayed in the Tiling Window reflects the duration of the corresponding tasks: the brighter an area is, the more time-consuming it is. On picture (a) we can distinguish the shape of the Mandelbrot set as depicted in Fig. 15. On picture (b), we observe that border tiles take a longer time to be processed than inner tiles.

its effectiveness by using the "*heat map*" mode[3] of the tiling window. In this mode, the brightness of tiles is proportional to the intensity of the associated computations. For illustration purposes, Figure 20a shows that, during the execution of the `mandel` kernel, the shape of the Mandelbrot set can be recognized at first sight right inside the tiling window. In the case of `blur` (Figure 20b), the tiling window reveals that inner tiles involve less computations than tiles located on the edges. This suggests that the code for inner tiles has been agressively optimized by the compiler.

Running EASYPAP in performance mode tells us that the gain achieved is beyond expectations: the new variant is 3 times faster! To analyze this performance boost, EASYVIEW offers a useful trace comparison feature, as shown in Fig. 21. We notice that many tasks are approximately 10 times faster than their original version. By moving the mouse over those tasks, students immediately get the confirmation that short durations do always correspond to inner tiles. The $\times 10$ speedup not only comes from the removal of conditional branches: it is mostly imputable to compiler auto-vectorization ($\times 8$ on `AVX2`-capable Intel processors).

Another interesting observation can be made when switching to the "*coverage map*" mode provided by EASYVIEW, using mouse horizontal mode to select all displayed tasks for a given CPU. In Fig. 21, the mouse cursor is over the CPU 3's timeline, so the purple squares displayed over the top-right thumbnail reveal the area covered by all tasks executed on this CPU during iteration range $[7..9]$. We observe that the squares are mostly regrouped in a single area, with only a few ones scattered in other places, which highlights the good locality property of the new `nonmonotonic:dynamic` scheduling policy.

[3]Heatmap mode can be toggled on/off by pressing the 'H' key

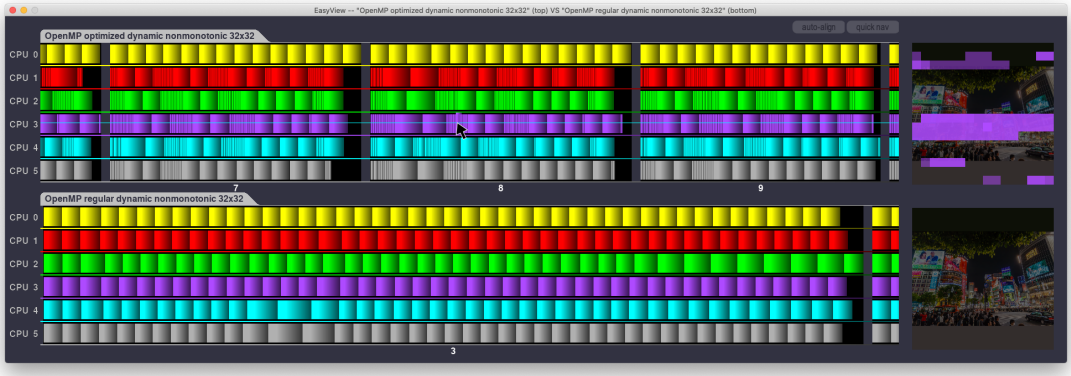


Figure 21: Comparison of two execution traces of the `blur` kernel using EASYVIEW. The bottom trace corresponds to the execution of a basic OpenMP implementation using uniform tiles. The top trace corresponds to an optimized OpenMP version where conditional code was removed from inner tiles. This later version is approximately 3 times faster in this setup (iteration 3 with the basic version is as long as iterations [7..9] with the optimized version).

Checking the correctness of the parallel variants can be done using the `--dump` option which stores the raw final image in a file. Students can thus use the `diff` Unix command to compare the output of a parallel variant against the output of the sequential one.

For students who want to go further, we propose to generate the longest tasks first (that is, the outer tiles) and then the shortest, which is a heuristic achieving good performance in practice. Care must be taken to use the `nowait` clause to avoid an implicit barrier between the two tasks waves: this is also something they can observe using EASYVIEW.

4.1.2. Identification of Connected Components

In more advanced courses, we introduce the students to the concepts of tasks and dependencies. After experimenting with OpenMP tasks on small programs, students are asked to parallelize a *Connected Components Detection* algorithm on 2D images. The main goal is to identify the different connected components (i.e. separated by transparent pixels) by coloring each of them in a unique color. The proposed algorithm first reassigns each pixel a unique color and then propagates the maximum between neighbours until reaching a steady state. Students are provided with a sequential implementation of the kernel which uses a sequence of two phases per iteration: the first phase propagates local maxima to the right and to the bottom, and the second one proceeds to an up-left propagation.

Parallelizing this algorithm without introducing extra iterations is quite challenging. A possible solution is to use a tiled implementation in which tiles are processed with some constraints: during the bottom-right phase (resp. up-left), a tile can be executed when its left and upper (resp. right and lower) neighbours have been completed. With OpenMP tasks, these constraints directly translate into task dependencies, as sketched in Fig. 22.

```
  for (int j = 0; j < NUM_TILES; j++)
    for (int i = 0; i < NUM_TILES; i++)
#pragma omp task depend(in: tile[i - 1][j], tile[i][j - 1])    \
                 depend(inout: tile[i][j])                     \
                 firstprivate(i, j)
      tile_down_right (i, j);
```

Figure 22: Snippet showing the implementation of the down-right propagation using OpenMP tasks with dependencies.

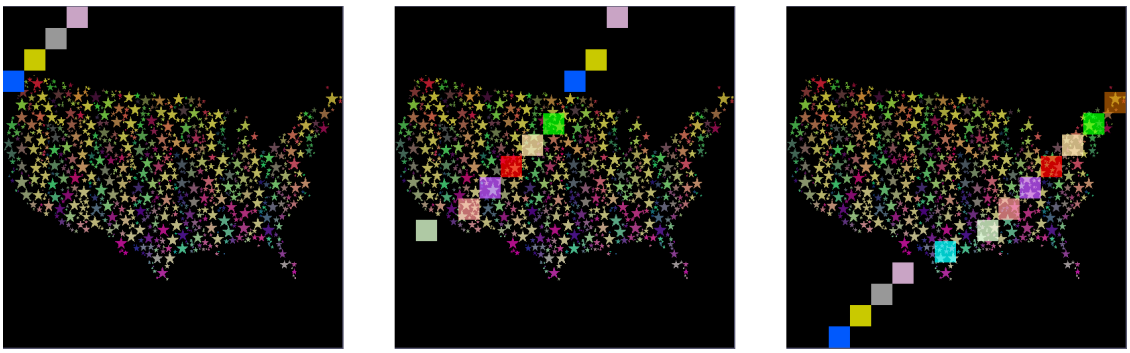


Figure 23: EASYVIEW allows to visualize the wave of tasks moving forward during the execution of code depicted in Fig. 22. These three screenshots were taken while moving the mouse from left to right over the Gantt window.

However, because it takes time to get familiar with the subtleties of task dependencies in OpenMP, students usually achieve a correct implementation only after several attempts. Most of the time, they over-constrain the problem and end up with a sequential execution of tasks. In such cases, EASYVIEW greatly helps to figure out if the dependencies were correctly enforced, as illustrated in Fig. 23. Students can observe the order in which tiles were processed by just moving the mouse.

4.2. GPU programming activities

GPU programming using closed-to-hardware interfaces (OpenCL, CUDA) is known to be difficult and error-prone. Fortunately, several concepts such as NVidia warps/AMD wavefronts or OpenCL workgroups can be illustrated graphically. We thus start our GPU practice sessions by making students experiment with very simple image manipulation kernels that they can slightly modify to visualize the coordinates of each workitem or the shape of the workgroups (e.g. by disabling workgroup with an even number).

We then use practice sessions which range from observation activities where students graphically explore advanced concepts such as *thread divergence* to more ambitious applications where students cope with the implementation of hybrid CPU-GPU computations requiring custom data structures and data exchanges between host memory and GPU GDDR memory at each iteration. We now present one example of each category.

4.2.1. Understanding divergence

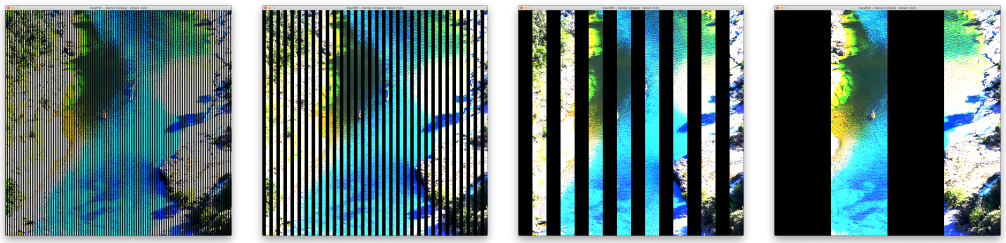
The notion of thread divergence, which is what happens when at least two threads do not take the same branch when performing a conditional jump in the code, is potentially harmful on a GPU. Due to the architectural design of modern GPUs, threads belonging to the same Nvidia warp (or AMD wavefront) are forced to execute the same intruction at each clock cycle, except if some are temporarily inhibited.

To further explore this constraint graphically, we provide our students with a `stripe` OpenCL kernel that lightens (resp. darkens) vertical stripes of a given image. The code is depicted in Figure 24. The kernel is executed with `DIM` $\times$ `DIM` workitems grouped in horizontal 1024×1 workgroups.

```
__kernel void stripes_ocl (__global unsigned *in,
                           __global unsigned *out, unsigned arg)
{
  int y = get_global_id (1);
  int x = get_global_id (0);

  if (x & arg)
    out [y * DIM + x] = brighten (in [y * DIM + x]);
  else
    out [y * DIM + x] = darken (in [y * DIM + x]);
}
```

Figure 24: OpenCL code of the `stripe` kernel. We assume that the `arg` parameter is a power of two (only a single bit to 1 in binary representation).

Students successively run the `stripe` kernel on a 1024×1024 image for different values of "`arg`" (1, 2, 4, 8, ...) to understand how it works. By looking at the EASYPAP main window (Figure 25), it appears obvious that the kernel draws vertical stripes (alternatively dark and bright) of width `arg`.



(a) arg = 4 (b) arg = 16 (c) arg = 64 (d) arg = 256

Figure 25: Result produced by the `stripe` kernel on a 1024×1024 image for different values of the `arg` parameter.

Since the code introduces a divergence (highlighted line in Fig. 24) between workitems calling `darken` and those calling `brighten`, students can now use EASYPAP in performance mode to look for the `arg` threshold value at which thread divergence is no longer harmful. This value should exactly correspond to the size of a *warp* (resp. a *wavefront*) on a NVidia (resp. AMD) GPU device.

Such experiments can easily be scripted, and students can then use Easyplot to obtain the plots reported

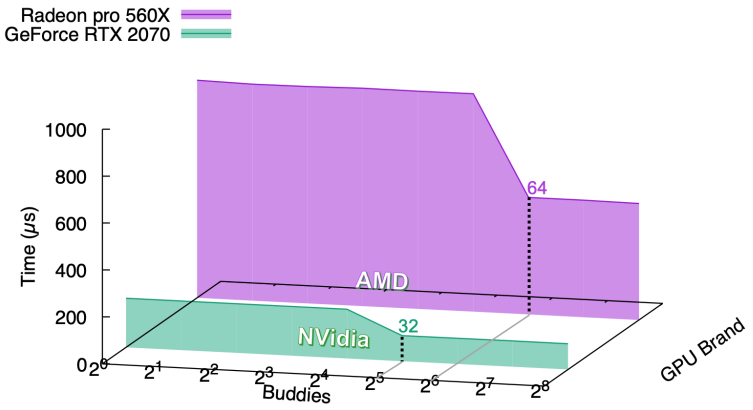


Figure 26: Execution time of the `stripe` kernel for different values of `arg`, on two different GPU architectures. In both cases, there is a threshold after which thread divergence is no longer harmful.

in Figure 26. In the end, they observe that a NVidia *warp* is formed by 32 threads, whereas an AMD wavefront is composed of 64 threads.

4.2.2. Hybrid OpenMP + OpenCL programs

Although most OpenCL assignments ask students to implement pure OpenCL kernels running on a single device, some more advanced assignments invite students to develop hybrid OpenMP + OpenCL programs where the GPU and the CPUs share the workload to address large datasets.

Let us take the Conway's *Game of Life* [13] assignment as an illustration. For the sake of simplicity, no load-balancing between the CPUs and the GPU is planned in a first phase: the frame is splitted into two equal horizontal rectangles. At each iteration, the CPUs work in parallel (using OpenMP) to compute the upper part of the frame while the GPU computes the lower part. Then, the *ghost cells* located at the frontiers must be exchanged (Game of Life is a 9-point 2D stencil code) before the next iteration can start: one contiguous line of pixels has to be transfered from host memory to the GPU, and its sibbling must be transfered in the opposite direction.

Once implemented, this hybrid kernel can be interactively observed to track the last bugs, and then the students can use EASYVIEW to observe the execution trace and make sure that computations on GPU and CPUs do overlap (Figure 27). They can also observe the cost of data transfers.

Motivated students often complete this assignment by adding a dynamic load-balancing mechanism which moves the frontier between the two rectangles so that CPUs and GPU complete each iteration at the same time.

Figure 27: The execution trace of a hybrid OpenCL + OpenMP version of the `life` kernel allows to make sure that the GPU and the CPU are working in parallel. Data transfers at the end of each iterations also appear in the GPU lane (red = host-to-device transfer, yellow = device-to-host transfer.

4.3. MPI distributed programming

In addition to Pthreads, OpenMP and OpenCL, EASYPAP also provides support for MPI programs, and most notably for debugging such programs using monitoring facilities. To illustrate this feature, let us take the *Game of Life* example again. This time, we ask students to develop a more elaborate "lazy" implementation that avoids recomputing tiles whose neighbourhood was in a steady state at the previous iteration. Once they end up with an effective Pthreads or OpenMP lazy variant, students can look at the tiling window to make sure that areas where "nothing changes" are not computed.

Finally, students extend their implementation in order to cope with distributed architectures by using MPI. Most of the difficulty lies in exchanging ghost-cells between MPI processes, including meta-informations regarding the state of tiles (steady or lively). The whole code is less than 150 lines, but is quite error-prone.

For debugging purposes, EASYPAP can display all the windows for each process, as previously discussed in Section 2.6. The following command launches two MPI processes executing the `mpi_omp` variant of the *Game of Life* kernel in debugging mode.

```
OMP_NUM_THREADS=4 easypap --kernel life --variant mpi_omp \
                          --mpirun "-np 2" --monitoring --debug M
```

The monitoring windows (Fig. 28) confirm that each process contains 4 threads and works on half of the image. Most importantly, since the sparse dataset consists in planers evolving along the diagonals of the image, we can check that only tiles located near diagonals are computed.

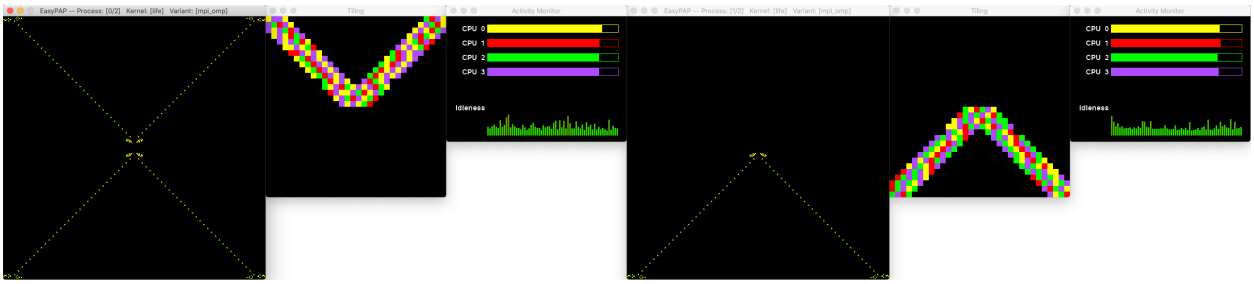


Figure 28: When launched in debugging mode, monitoring windows of every MPI process show up and help to visualize which area is processed by each of them.

5. Pedagogical considerations

Since 2018, we have progressively introduced EASYPAP in our lab sessions in the context of two courses:

- **System Programming** which is given during the 3rd year of university. This course is about Unix programming environment (files, processes, pipes, signals, non-local jumps, threads). EASYPAP is used during two labs to introduce students to HPC (Posix thread, basic concepts of OpenMP). Third year students have some good programming knowledge (C, python and CAML) and algorithmic skills (mainly data structures). They are familiar with computer architecture, assembler language and network programming.

- **Parallel Architecture Programming** which is given during the 4th year of university. Students have a solid programming and algorithmic knowledge, they have good skills in compilation, operating system and complexity theory. We both study advanced hardware architecture (instruction level parallelism, thread level parallelism, data parallelism) and parallel programming (advanced OpenMP, OpenCL, basic concepts of MPI). EASYPAP is used during 8 labs and students have to achieve a significant project: a kind of *putting it all together* assigment where they are asked to produce several distinct parallel variants and study their behaviour on different initial datasets and different hardware. Some datasets lead to a uniform workload while others involve a very irregular workload. Students implement advanced OpenMP variants, vectorized variants and a basic OpenCL or MPI variant. They are also encouraged to produce hybrid variants (e.g. mixing MPI and OpenMP). Finally, they have to write a scientific report explaining and highlighting their work.

In the remaining of this section, we report on our teaching experience in the context of the Parallel Architecture Programming Course. First we use Bloom's taxonomy [7] to classify some of our pedagogical activities in order to demonstrate the usefulness of EASYPAP at all levels of learning. We also present students' responses to an anonymous survey about the relevance of EASYPAP as a software and as a pedagogical tool.

5.1. Bloom's levels of understanding

We believe that the EASYPAP environment is a good tool to engage learners not only because it proposes fun features but it can be usefull to master new knowledge at different *levels of understanding*. By levels of understanding we refer to the famous taxonomy of Bloom [7], published in 1956, which is a way to categorize the levels of reasoning skills required in classroom situations. Bloom has defined six levels in this taxonomy, each requiring a higher level of abstraction from students: knowledge, comprehension, application, analysis, synthesis end evaluation. Following Bloom, a group of cognitive psychologists and educational researchers published in 2001 a revision of Bloom's Taxonomy in [3] in order to integrate modern developments. For instance, students are now considered more responsible for their own learning, cognition and thinking. Therefore nouns of the original Bloom's taxonamy were replaced by verbs in the revisited taxonomy. Interestingly, the order of Synthesis and Evaluation levels has been reversed: in a way, creativity is now considered more important than the ability to pass a judgment. This new taxonomy is presented in the form of a two dimensionnal table: the *cognitive process* dimension and the *knowlegde* dimension. However, for the sake of simplicity only the cognitive process dimension is usually used. We classify in Table 1 some of our activities with EASYPAP according to Bloom's taxonomy. We can note that lectures and labs mainly focus on the first 4 levels whereas the levels *Evaluate* and *Create* are specifically worked during *putting it all together* assigment.

5.2. Students' feedback

In March 2020 we conducted an anonymous online survey about EASYPAP, just prior to France covid'19 lockdown measures. The surveyed population consisted of 59 University computer science students (postgraduate level) 28 engineering school students. All students had a record of about eight hours of practice with EASYPAP. The *putting all together* assigment was given during the lockdown period, so the survey does not cover this homework activity. The purpose of the survey was clearly indicated to students: the goal was to collect "objective" data in order to publish them in a scientific article. We gathered answers from 27 Master students and 14 engineering students, representing a response rate of 47%. The survey was divided into three parts, the first one was about the EASYPAP software, the second one was about the pedagogical interest of using EASYPAP and the last part was dedicated to open-ended comments. We used 5-point Likert scale questions, however the answers *strongly disagree* and *strongly dissatisfied* were never checked.

The summary of survey responses about EASYPAP as a software is shown in Fig. 29. Students are generally satisfied with the software. However, some students have experienced difficulties in installing it at home during the lockdown period. Most issues were related to finding an appropriate OpenCL driver compatible for their custom installation, which is admittedly often challenging. Nevetheless, after a brief period of panic, all students were able to use either their personal computer or University's servers remotely

Cognitive Process	Student's engagement activities with the help of EASYPAP
Remember *Retrieving relevant knowledge from long-term memory*	As we mentioned in Section 3.1 using EASYPAP involves increased use of long-term visual memory for some factual notions (eg. granularity, load balancing) and for more conceptual knowledge like bad behavior recognition as discussed in Section 3.3.
Understand *Determining the meaning of instructional messages. Including oral and graphic communication.*	During lectures and labs we use EASYPAP with different codes to illustrate concepts of parallelism (section 3.1) and to observe the concrete meaning of keywords (Section 3.2).
Apply *Carrying out or using a procedure in a given situation.*	As we have mentioned in section 4, we have developed a whole set of assignments ranging from the basic application of lectures to the implementation of OpenMP, MPI or OpenCL variants. Student productivity during labs is improved because a single environment is used and students can check their work visually.
Analyse *Breaking material into constituent parts and detecting how the parts relate to one another and to an overall structure or purpose.*	Students are asked to optimize kernels that have poor parallel behaviors. Kernels may be given by the teacher or be produced by students themselves. To do this, students have to (1) identify bad performance issues thanks to trace analysis, (2) make the link between an issue and the source code and (3) modify the code to improve it. Examples of such optimization are also given during lectures and in the tutorial presented in section 3.4.
Evaluate *Making judgment based on criteria and standards.*	Students have to write a report to present a *putting it all together assigment*. In this report they must scientifically compare optimizations using performance graphs: (1) select data to produce meaningful performance graphs, (2) explain the role of parameters through evidence like traces and facts like the behavior of computer hardware. A tutorial presented in section 3.4 is given to students to guide them through this activity.
Create *Putting elements together to form a novel, coherent whole or make an original product.*	For the *putting it All Together assigment* students are asked to design new parallel algorithms (and data structures) and to write a scientific report.

Table 1: Classification of some educational activities with EASYPAP according to Bloom's taxonomy.

to perform their experiments. To this end, we added several features to EASYPAP to ease its utilization through a remote connection.

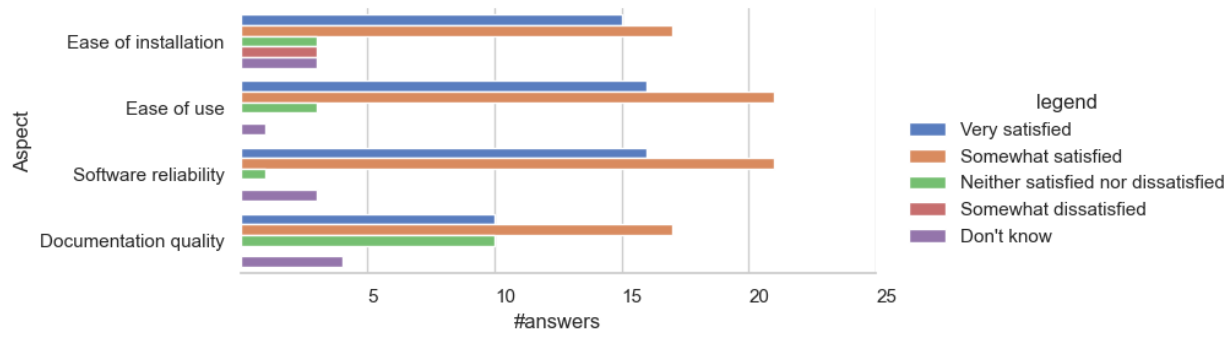


Figure 29: Software quality of EASYPAP

We also tried to figure out how students would describe the pedagogical impact of the EASYPAP approach on their motivation, understanding, productivity and creativity. For this purpose, we have clarified these different notions as follows:

Motivation Seeing demonstrations of easypap makes you want to learn more about parallel programming - using easypap encourages you to work during labs.

Understanding of concepts Using EASYPAP makes it easier to understand (i.e. to perceive by the mind, by reasoning) the concepts of parallel programming (load balancing, granularity of computation, management of shared data, need for synchronization mechanisms).

Technical understanding Using EASYPAP facilitates the understanding of parallel machines, parallel programming environments and parallel programs.

Productivity Using EASYPAP allows you to go straight to the point during practical work without wasting time - using easypap saves you time (bug discovery, undesirable behaviors).

Creativity Using EASYPAP has given you ideas (optimizations during practical work, personal programming project).

Responses to the survey on the pedagogical impact of EASYPAP are presented in Fig. 30. Clearly, most students acknowledge that EASYPAP has helped them to better understand parallelism and has encouraged them to spend more time on their assignment. The results on creativity are more mixed. However, as already mentionned, the students did not had a chance to start their *putting all together* assigment at that time, where creativity is specifically addressed. Our reading is that 18 students out of 41 felt to be really engaged during the lab sessions and, on the opposite, 4 students did not.

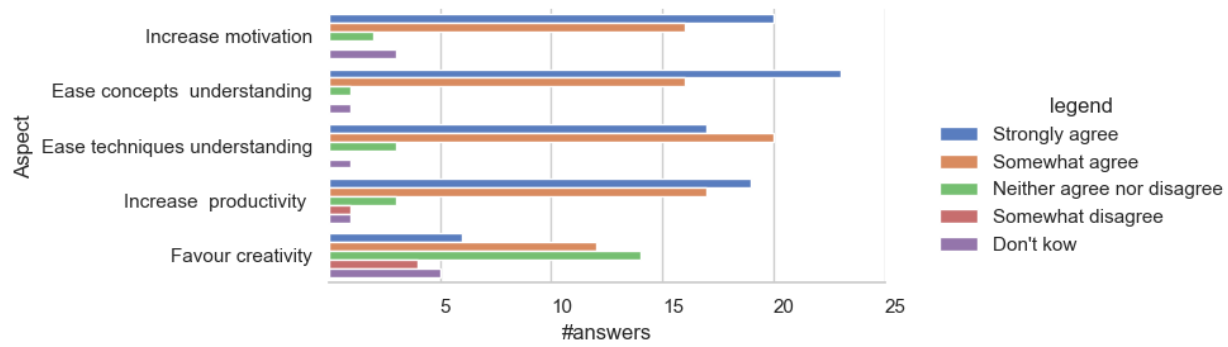


Figure 30: Impact of EASYPAP on pedagogy

The last part of the anonymous survey was devoted to open-ended comments. We received 9 comments in which the students specified the points they preferred, mainly the motivation brought by EASYPAP and the fact of being able to mostly focus on parallelism concerns. We also received some relevant remarks. In particular, a student suggested that we provide EASYPAP in the form of a dynamic library so that it could be used to parallelize existing applications.

5.3. Instructors' feedback

Since we introduced EASYPAP in our lectures and lab sessions, it is clear from our side that students find parallel programming much more attractive and fun. As we already mentioned, EASYPAP is our great companion during parallel programming lectures. It injects dynamism in our lectures by allowing us to progressively explore new concepts using an interactive *test-fail-retry* approach.

During labs, graphical tools make student debugging sessions less painful and more effective. The fact that it took postgraduate students less than a dozen hours to come up with an efficient MPI+OpenMP implementation of the *Game of Life* kernel using lazy evaluation (see Section 4.3) is a tangible improvement. This highlights the importance of allowing students to quickly prototype preliminary variants of the code and analyze their parallel behavior interactively. However, we have to temper this last point. Indeed we use a script to help us grading the projects: for each variant provided by the student this script checks the obtained image against the expected image and automatically calculates the speedup. However, we were surprised to find that half of the students had submitted at least one buggy version even though they had the tools to check it ! For the coming years we are thinking of implementing an automatic grading system to save time for the students and improve the quality of the submitted assignments like it was successfully deployed at Rice University [15]. Moreover we will organize a performance contest to encourage emulation between students or, even better, apply gamification to our course [12, 14].

On the downside, EASYPAP provides the students with an integrated environment where all the low-level details of configuration, compilation and initialization of various components are hidden. So it is

necessary to conduct more conventional lab assignments as well, involving writing small applications from scratch to show students how a complete OpenCL program looks like for instance.

Finally, we –as teachers – have learned a lot using EASYPAP. We had the opportunity to observe behaviors that would normally be cumbersome to look at. For instance, by comparing the execution traces generated with different compilers (namely gcc and clang), we observed very different strategies regarding the order in which the tasks get executed. Also, the impact of cache on performance is always tricky to predict, and by looking at execution traces, we became aware that cache-prefetching never crosses operating system's page boundaries.

6. Related Work

There have been many contributions to the field of developping programming environments for teaching parallel programming [5, 8, 6].

Like the authors of [5], we are convinced by the pedagogical benefits of using *exemplars*. EASYPAP is also built around the notion of exemplars that students can parallelize using multiple paradigms.

Regarding visualization, we adhere to the same philosophy as the ParaVis [8] and TSGL [6] efforts, which provide easy-to-use C/C++ interfaces to visualize 2D animations produced by parallel computations. These libraries are versatile and can be interfaced with almost any existing 2D simulation. EASYPAP follows a different approach by providing an integrated educational framework with monitoring and trace exploration capabilities, experience automation and plot generation assistance.

Many outstanding tools have been developed to visualize and analyze execution traces, such as Aftermath [10], Grain Graphs [18], Intel Vtune Profiler [1], TAU [20], Vampir [17] or ViTE [4]. We think EASYPAP represents a smooth and attractive first contact with trace analysis tools, before beeing introduced to more complex ones. An original aspect of both EASYPAP and EASYVIEW is that they establish a graphical link between computations (i.e. tasks) and their associated data (i.e. image tile).

7. Conclusion and Future Work

EASYPAP is a framework designed to make learning parallel programming more accessible and attractive to students. A comprehensive set of tools allows to quickly get visual feedback about the parallel behavior of their code, to analyze the locality of the computations, and to understand performance issues.

The use of EASYPAP for two years, in the context of undergraduate and postgraduate courses on parallel programming at University of Bordeaux, was very successful. Students were able to understand very subtle aspects of scheduling, synchronization and compiler optimizations. We have also used EASYPAP to popularize parallel programming for middle school students. It made it possible to easily illustrate concepts such as load imbalance.

In a near future, we plan to release EASYPAP in the form of a separate library, allowing existing applications to take advantage of the visualization, monitoring and tracing facilities of our platform. We also intend to further extend the EASYVIEW trace explorer to integrate per-task cache usage information using the PAPI library [22].

References

[1] "Intel Vtune Profiler," (Visited on 2020-03-09). [Online]. Available: https://software.intel.com/en-us/vtune

[2] "SDL: Simple directmedia layer," (Visited on 2020-03-09). [Online]. Available: https://www.libsdl.org

[3] *A taxonomy for learning, teaching, and assessing : a revision of Bloom's taxonomy of educational objectives / editors, Lorin W. Anderson, David Krathwohl ; contributors, Peter W. Airasian ... [et al.].*, Complete ed. ed. New York: Longman.

[4] "ViTE: Visual trace explorer," (Visited on 2020-03-09). [Online]. Available: http://vite.gforge.inria.fr

[5] J. Adams, R. Brown, and E. Shoop, "Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to cs undergraduates," in *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum*, 2013.

[6] J. C. Adams, P. A. Crain, and M. B. V. Stel, "Tsgl a thread safe graphics library for visualizing parallelism," *Procedia Computer Science*, vol. 51, pp. 1986 – 1995, 2015, international Conference On Computational Science.

[7] B. Bloom, M. Engelhart, E. Furst, W. Hill, and D. Krathwohl, "Taxonomy of educational objectives: The classification of educational objectives," vol. 1, 01 1956.

[8] A. Danner, T. Newhall, and K. Webb, "Paravis: A library for visualizing and debugging parallel applications," in *9th NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-19)*, 2019.

[9] A. Douady and J. Hubbard, "Exploring the mandelbrot set. the orsay notes." 10 2009.

[10] A. Drebes, J.-B. Bréjon, A. Pop, K. Heydemann, and A. Cohen, "Language-centric performance analysis of openmp programs with aftermath," in *OpenMP: Memory, Devices, and Tasks*. Springer International Publishing, 2016.

[11] ETP4HPC, "Strategic research agenda," 2017, (Visited on 2020-03-09). [Online]. Available: https://www.etp4hpc.eu/pujades/files/SRA%203.pdf

[12] J. Fresno Bausela, H. Ortega-Arranz, A. Ortega-Arranz, A. Gonzalez-Escribano, and D. Ferraris, *Applying Gamification in a Parallel Programming Course*, 01 2017.

[13] M. Games, "The fantastic combinations of john conway's new solitaire game "life" by martin gardner," *Scientific American*, vol. 223, pp. 120–123, 1970.

[14] A. Gonzalez-Escribano, V. Lara-Mongil, E. Rodriguez-Gutiez, and Y. Torres, "Toward improving collaborative behaviour during competitive programming assignments," in *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, 2019, pp. 68–74.

[15] M. Grossman, M. Aziz, H. Chi, A. Tibrewal, S. Imam, and V. Sarkar, "Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level," *Journal of Parallel and Distributed Computing*, vol. 105, pp. 18 – 30, 2017, keeping up with Technology: Teaching Parallel, Distributed and High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517300047

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[17] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, pp. 139–155.

[18] A. Muddukrishna, P. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: Openmp performance analysis made easy," *ACM SIGPLAN Notices*, vol. 51, pp. 1–13, 02 2016.

[19] R. Namyst and P.-A. Wacrenier. (2018) The EASYPAP web site. (Visited on 2020-03-09). [Online]. Available: https://gforgeron.gitlab.io/easypap/

[20] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[21] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[22] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, 2010, pp. 157–173.

[23] M. Waskom and the seaborn development team, "mwaskom/seaborn," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.592845