EASYPAP: a Framework for Learning Parallel Programming

Alice Lasserre, Raymond Namyst, Pierre-André Wacrenier

EASYPAP: a Framework for Learning Parallel Programming

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

[a] *University of Bordeaux, INRIA, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, 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 with this many-core trend which has impacted every commodity hardware, from smartphones to desktop machines. To get the most out of today's computers, people must be trained in parallel programming. Thus, 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 [12].

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 non-determinism 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 framework 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, SIMD intrinsics, 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. Section 2 presents how we teach parallel programming at University of Bordeaux and explains why we have designed a new pedagogical framework. The EASYPAP environment and its associated tools are described in Section 3. In Section 4, 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 detailed in Section 5 and a long term project is described in Section 6. We discuss how our pedagogical approach fits in Bloom's taxonomy and present an evaluation of EASYPAP in Section 7. We position our approach with respect to related work in Section 8. Finally, we give concluding remarks in Section 9.

*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)

2. Background and Motivation

At the University of Bordeaux, most topics related to parallel programming are part of the Computer Science Master's curriculum. Nonetheless, undergraduate students are being taught thread programming in the context of operating systems, as a way of running loosely coupled activities concurrently. When they start their Master's degree, students are thus familiar with the concepts of asynchronous execution, race conditions and synchronization tools (semaphores and Hoare Monitors).

In the Master's curriculum, the *Parallel Programming* course is the course where students are introduced to high performance computing. The topics covered by this course are detailed in Table 1. The *Parallel Programming* course spans over eleven weeks. Each week, students attend a two-hour lecture and have a two-hour practice session on high-end workstations[1].

Theme	Covered Topics
Introduction	Load balancing problem, Amdalh's law, fork/join model, OpenMP basics.
Expressing parallelism	Loop parallelism and scheduling strategies for regular and irregular computations, nested parallelism for divide and conquer algorithms, graph of dependent tasks for specific applications.
Memory considerations	Cache-conscious and cache oblivious algorithms, memory allocation and thread mapping, false sharing.
Data-parallelism	CPU architecture, vector processing, compiler auto-vectorization, SIMD intrinsics.
GPU Programming	GPU architecture, OpenCL, Thread divergence, memory coalescing, reductions.
Distributed Computing	Introduction to MPI with iterative stencil loops.

Table 1: Topics covered by the Parallel Programming course offered by the Computer Science Master's Curriculum at the University of Bordeaux

Our approach to teaching parallel programming is based on the following three principles. First, we engage the students in hands-on activities for every topic addressed in the course. Students generally start experimenting with a single topic at a time, and they later are assigned wrap-up exercises where they learn how multiple aspects may interfere within more complex codes. For instance, when they are introduced to hardware caches, they observe the influence of data layout and data traversal strategies on the cache hit ratio, then they experiment with the cache prefetcher, then they observe the effects of false sharing in the context of multithreaded programs, and they end up putting it all together by optimizing a memory-bound parallel kernel. Second, we strive to design pedagogical materials which allow students to interactively observe the graphical output of each code. Like many other colleagues [8, 2, 19], we think that visualization is the key to better understand and experiment with new concepts. We have learned from our past experiences that providing visualization facilities is always worth the effort, even for low-level, hardware-specific aspects for which it may not be straightforward to find a graphical illustration. Visualization is also a time-saver for students as it often leads to quicker detection and understanding of erroneous behavior of their parallel variants. Moreover, using animated graphical outputs increases students' motivation during lab sessions. Third, we train students to adopt a scientific approach while exploring the different optimizations they gradually apply to their code. They are encouraged to conduct thorough experiments to study the influence of each parameter and to track down potential sources of inefficiency.

Year after year, we have consolidated a database of 2D-enabled applications which did cover almost all topics taught during our parallel programming lectures. However, our database could be significantly improved. Because these applications where not sharing a common software core nor the same command-line interface, becoming operational could take some time to students when switching to a new assignment. Moreover, understanding performance issues was still a big challenge, because instrumenting and profiling parallel codes is often tedious, and using full-featured off-line trace analyzers is complex for beginners.

In the light of these observations, we have designed EASYPAP, a framework which allows students to experiment with various parallel paradigms (Pthreads, OpenMP, SIMD, OpenCL and MPI) through a uniform environment. EASYPAP integrates a real-time activity monitor and an off-line trace analyzer that provide new ways of visualizing tasks together with their associated data. Students can not only

[1] 24-core machines equipped with recent graphics cards (CPU : 2× Xeon Gold 5118 64Go, GPU : NVIDIA RTX 2070)

diagnose common performance issues faster, but they can also link the problem to the data manipulated.

3. The EASYPAP Framework

EASYPAP is a C programming environment that relies on the SDL library [32] 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 [23]. 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. When doing their practice work with EASYPAP, our students typically adopt the following workflow. They first start by implementing a first prototype and they visually check its correctness at first glance. Then, they observe the overall parallel behavior with real time monitoring tools, and inspect the impact of various parameters on the scheduling of computations. Once the behavior looks satisfactory, they perform experiments with multiple parameters to estimate the quality of the achieved speedup. To investigate how the speedup could be further improved, or to diagnose performance issues, they use an interactive, off-line trace analyzer to detect scheduling issues, poor task implementation or even memory access conflicts. They end up with multiple variants of the code that they can thoroughly compare and analyze using plotting facilities.

3.1. 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 prefixed by `kernelname_compute_`.

As an illustration, consider the simple `spin` kernel, the sequential implementation of which is shown in Fig. 1. This kernel colors the pixels of an image according to their polar coordinates. 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 across multiple iterations. 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. This variable can be changed to refresh the display only every `nb_iter` iterations. When running in performance mode (see Section 6.4), no display is involved and this variable is set to the total number of iterations requested by the user.

```
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`

Lines 4–6 illustrate how the contents of the image are accessed during an iteration. For the sake of simplicity, EASYPAP works on square-shaped images. Each pixel can be 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 displays 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. EASYPAP uses the `dlsym` Unix dynamic linker facility to find and call the appropriate function (i.e. `spin_compute_seq`).

Given that 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`" directive before the **`for`** loop iterating over lines, recompile EASYPAP, and relaunch using `--variant omp`. The obtained graphical animation allows the students to visually check if this variant produces the expected output and if it runs faster. In a second step, students will need to accurately benchmark and compare the performance of multiple variants. To this end, when invoked with the `--no-display` option, EASYPAP eliminates the overhead of graphical updates and simply reports the overall wall clock time after completion of the requested number of iterations.

The simplicity with which students are able to implement many different variants of each kernel is an essential feature of EASYPAP. The fact that all parameters can be passed as command line arguments makes it very convenient to compare variants against each other, either interactively or using automation scripts, as we further discuss in the next sections.

3.2. Online monitoring

```
// 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
    rotate ();
  }
}
```

Figure 2: Typical example of instrumented code using calls to `monitoring_start_tile` and `monitoring_end_tile`. To ease the use of tiling, EASYPAP provides two global variables `NUM_TILES` and `TILE_SIZE` which may be set using command line options. Value of the unspecified variable is deduced using the formula `DIM` = `NUM_TILES` × `TILE_SIZE`.

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`. 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 a given rectangle. Note that while our example only uses square tiles, in the general case any rectangular shape is acceptable. 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()`.

Once the code is instrumented, real-time monitoring can be activated using the `--monitoring` option. This 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 contrast with system-wide performance monitors, 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 *accumulated 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*

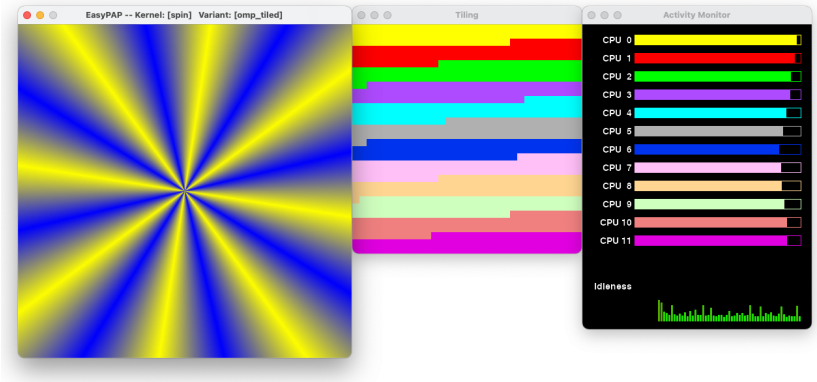
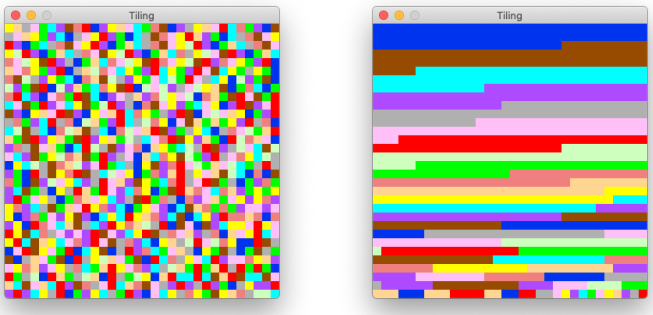


Figure 3: In addition to the main graphical kernel output, the monitoring mode introduces two additional windows: a tiling window (top) and a CPU monitoring window. This screenshot was obtained during the execution resulting from the following command: `easypap --kernel spin --variant omp_tiled --tile-size 64 --monitoring`

window. By observing Fig. 3, we see that the tiles have been assigned to threads in contiguous chunks, in accordance with the *static* loop scheduling policy.

For teachers, the tiling window is also a useful tool to graphically illustrate 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 `dynamic` policy, whereas Fig. 4b shows how the size of chunks assigned to threads decreases over time with the `guided` policy.



(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.

3.3. 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 can ask EASYPAP to record tile-related profiling events (i.e. start/end time, tile coordinates, CPU) into a trace file. To visually explore and interact with the collected trace, we have designed the EASYVIEW utility (Fig. 5). 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 image computed at the selected iteration (see thumbnails of the spinning wheel appearing in Fig. 5). 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 the mouse smoothly 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 the selection of a particular CPU and highlights the tiles computed during the displayed period. Selecting a CPU allows them to observe the "coverage map" of a the CPU during one or multiple iterations, and to check the locality of computations across iterations.

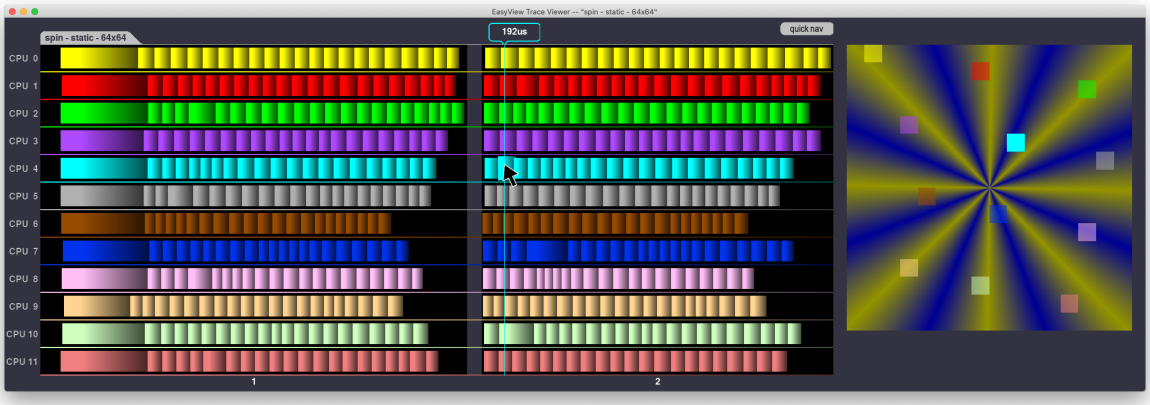


Figure 5: EASYVIEW brings interactive exploration of traces generated by EASYPAP. Moving the mouse over a task in the Gantt diagram displays its duration (bubble at top of window). This trace shows the first two iterations of the OpenMP tiled variant of `spin`. Most tasks have a homogeneous duration (around 200μs) with the notable exception of the first tasks computed by each CPU, which do not benefit from the "*warm cache*" effect.

EASYVIEW is a powerful means 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 5.

3.4. GPU Programming with OpenCL

Although directive-based high-level languages such as OpenMP [25] or OpenACC [24] allow programmers to seamlessly offload code regions to accelerators, we think that learning GPU architecture together with its associated low-level execution model is the key to understand GPU programming. In the context of our *Parallel Programming* course, we have thus decided to use the OpenCL [29] environment which supports a wide range of accelerators.

The integration of OpenCL into our EASYPAP framework 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, so students don't have to cope with the discovery of hardware platforms, the creation of contexts, buffers and queues, and the compilation of kernels.

As for CPU kernels, a buffer named `cur_buffer` is pre-allocated on the device and the initial image (if any) is transferred to this buffer before the computation loop starts. Figure 6 shows the contents of the `spin.cl` file which features the definition of the 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` work-items (which is the default behavior of EASYPAP), each work-item being responsible for computing only one pixel (Fig. 6, lines 9–11).

```
1  static unsigned compute_color (int y, int x, const float drawing_angle)
2  {
3    ... // basically a copy/paste of the sequential code
4  }
5
6  __kernel void spin_ocl (__global unsigned *out,
7                          const float drawing_angle)
8  {
9    int x = get_global_id (0), y = get_global_id (1);
10
11   out [y * DIM + x] = compute_color (y, x, drawing_angle);
12 }
```

Figure 6: OpenCL implementation of the `spin` kernel

On the CPU side, the students are provided with a generic "kernel invocation routine" that they can customize to pass appropriate parameters to their implementation. In the case of the `spin` kernel, students end up with the function depicted in Figure 7. The code presented in Figures 6 and 7 faithfully represents all students need to write to obtain an effective `spin` OpenCL implementation. OpenCL kernels variants are launched the same way[2] as any CPU variant, with most parameters applying to both targets. Using a consistent command line interface is particularly useful when using scripts to automate experiments. More importantly, students feel more comfortable when going from OpenMP to OpenCL programs: they are instantly able to run interactive experiments, plot curves, observe monitoring data or inspect execution traces.

```
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 7: OpenCL Launcher function responsible for invoking the `spin` kernel. Highlighted lines illustrate how OpenCL code can easily be instrumented for monitoring purposes.

Like OpenMP kernels, the execution of OpenCL kernels can be instrumented for monitoring or trace recording purposes. Code highlighted in Figure 7 illustrates how the timing events related to each kernel execution can be easily captured. Note that this GPU instrumentation is a coarse grain one, since we have no detail about what happens inside the device (e.g., at the workgroup level). Nonetheless, observing the activity periods of an OpenCL device is always useful, especially when the device is 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 in order to appear in the execution trace. This feature is further explored in Section 5.2.

3.5. 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 will be illustrated later). Once initialized, all MPI processes execute the `spin_compute_mpi` function in a SPMD[3] 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 8 illustrates how a MPI-variant of the `spin` can be implemented. Because the computation load is the same for every pixel of the image, we use a block-decomposition method such that each process is in charge of a distinct horizontal stripe of the image. The bounds of the stripe assigned to the current process are determined in the `spin_init_mpi` initialization function (line 5–12). The actual computation of the stripe inside each process is parallelized using OpenMP (line 17). Eventually, the contributions of all processes are brought back on the master process using a `MPI_Gather` collective operation (lines 21–22).

[2]The only minor difference is that the `--ocl` flag must be passed to activate OpenCL-specific initializations.
[3]Simple Program Multiple Data

```
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    MPI_Comm_rank (MPI_COMM_WORLD, &mpi_rank);
8    MPI_Comm_size (MPI_COMM_WORLD, &mpi_size);
9
10   mpi_y = mpi_rank * (DIM / mpi_size);
11   mpi_h = (DIM / mpi_size);
12 }
13
14 unsigned spin_compute_mpi (unsigned nb_iter)
15 {
16   for (unsigned it = 1; it <= nb_iter; it++) {
17     do_tile_openmp (0 /* x */, mpi_y /* y */, DIM /* width */, mpi_h /* height */);
18     rotate ();
19   }
20
21   MPI_Gather ((mpi_rank == 0 ? MPI_IN_PLACE : image + mpi_y * DIM), mpi_h * DIM,
22               MPI_INT, image, mpi_h * DIM, MPI_INT, 0, MPI_COMM_WORLD);
23
24   return 0;
25 }
```

Figure 8: 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. Data transfers are performed only when all the pixels need to be gathered on the master process (lines 21–22).

When running EASYPAP in interactive mode, only the graphical window of the master MPI process is displayed by default. However, for debugging purposes, the user can activate the display of every process's 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 9 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 works 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, as a result of the `MPI_Gather` operation.

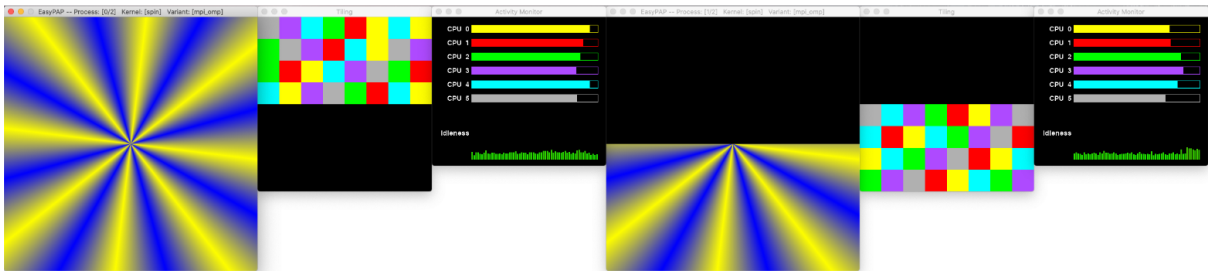


Figure 9: Snapshot of a MPI+OpenMP variant of the `spin` kernel. The execution uses two MPI processes, each hosting 6 OpenMP threads.

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 visualized with EASYVIEW, either individually or using a side-by-side comparison to diagnose load balancing issues for instance.

4. Transmitting knowledge about HPC with EASYPAP

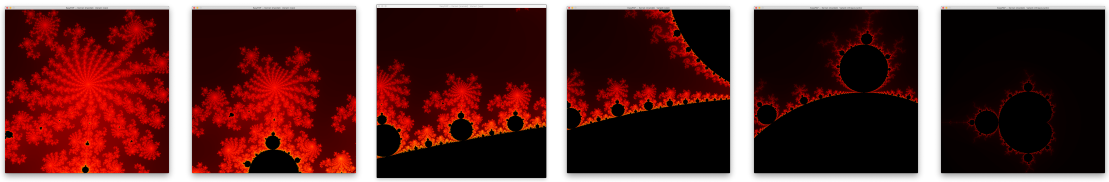
Although EASYPAP was primarily designed as a practice framework to help student to implement and experiment with parallel computations, it proved to be also a great teachers' companion during *parallel programming* lectures, in a pure "transmissive" teaching mode.

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. We use EASYPAP during our lectures to motivate graduate students either by showing a preview of what they will achieve during lab sessions, or by demystifying advanced language constructs in a graphical manner (e.g., behavior of the new OpenMP 5 *nonmonotonic* scheduling policy). Moreover, EASYPAP 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.

4.1. Popularizing the basic concepts of parallelism

The computation of the Mandelbrot set [10] 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 10.

When running the sequential variant of `mandel` in interactive mode, the animation plays quite smoothly during the very first iterations, but then 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 10b). This slowdown goes worse as the Mandelbrot set grows and occupies a bigger portion of the frame (from iteration 30 to approximately 200 in Figure 10). Note that even if students are not familiar with Mandelbrot 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 10: 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, each stripe being assigned to a distinct CPU (Figure 11). At this point, showing the (OpenMP) source code is not needed. We only want students to observe the acceleration in comparison with the previous sequential run. In order to draw attention to the way the workload was assigned to CPUs, we run EASYPAP under monitoring mode (Figure 11). 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 11) clearly reveals a strong load imbalance between CPUs. This issue is further confirmed by the "*heat map*" mode of the tiling window, a display mode where the brightness of tiles is proportional to the intensity of the associated computations. 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 many more computations than other areas.

At this point, an interesting discussion can take place with the audience about what would be a good work distribution strategy. Even if students have no specific knowledge about existing OpenMP

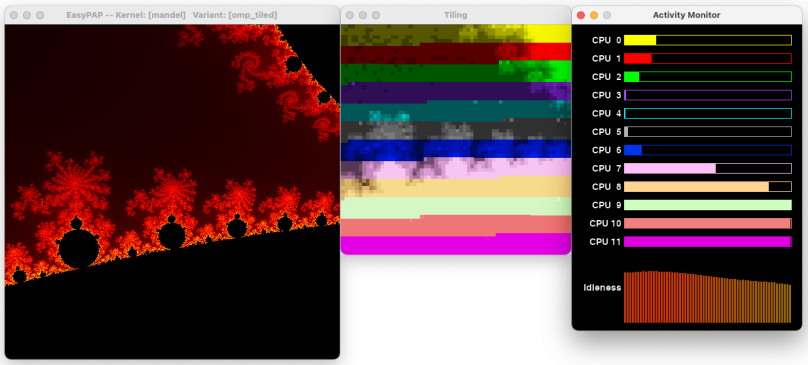
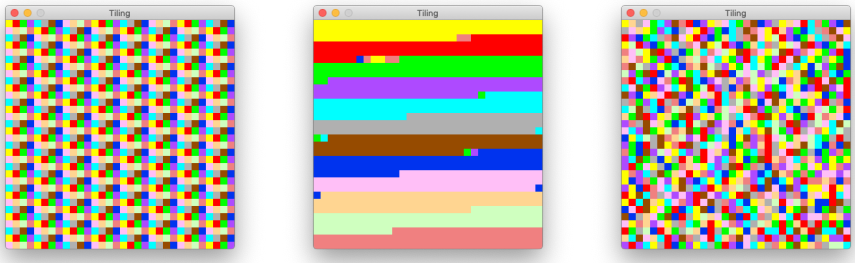


Figure 11: Screenshot taken after 100 iterations of the OpenMP variant of the `mandel` kernel with a static distribution of tiles. The *heat map* monitoring mode was used, so that the brightness of tiles displayed in the Tiling Window reflects the duration of the corresponding tasks: the brighter an area is, the more time-consuming it is.

scheduling policies, the discussion quickly focuses on fairer ways of assigning small tiles of the frame, such as cyclic or dynamic distributions.

4.2. Understanding work distribution policies

As discussed in Section 3.2, the tiling window is particularly useful for observing and comparing different work distribution policies. Similarly to the `spin` kernel (Section 3.2), our OpenMP variant of `mandel` uses a tile-based decomposition, the assignment of tiles being customizable at run time. Changing the loop scheduling policy is thus a matter of setting the `OMP_SCHEDULE` environment variable to the desired policy. Figure 12 shows the output of the tiling window when using three different loop scheduling policies.



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

Figure 12: The Tiling Window allows to compare the different OpenMP loop scheduling policies on the actual distribution of tiles.

On Subfigure 12a, 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 significantly improves performance over the block-static distribution because "heavyweight" tiles are somewhat homogeneously distributed among CPUs.

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. However, as soon as some threads go idle, they steal work from overloaded threads. Work stealing is clearly visible in Subfigure 12b, 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 12c) further enhances performance, as distributing the tiles in a greedy 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. 13.

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

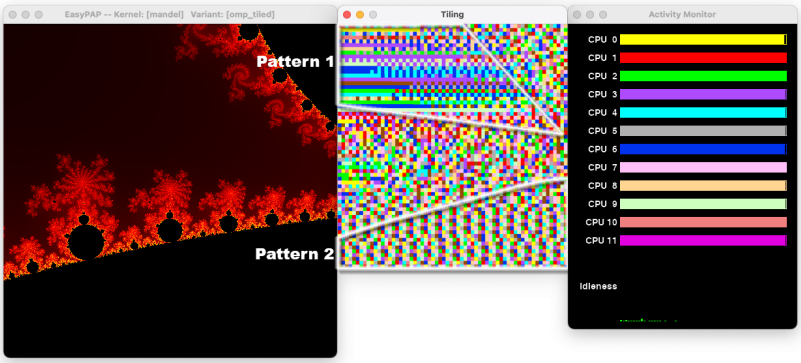


Figure 13: When using OpenMP dynamic loop scheduling of small tiles, the tiling window reveals two noticeable patterns.

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.

4.3. 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.

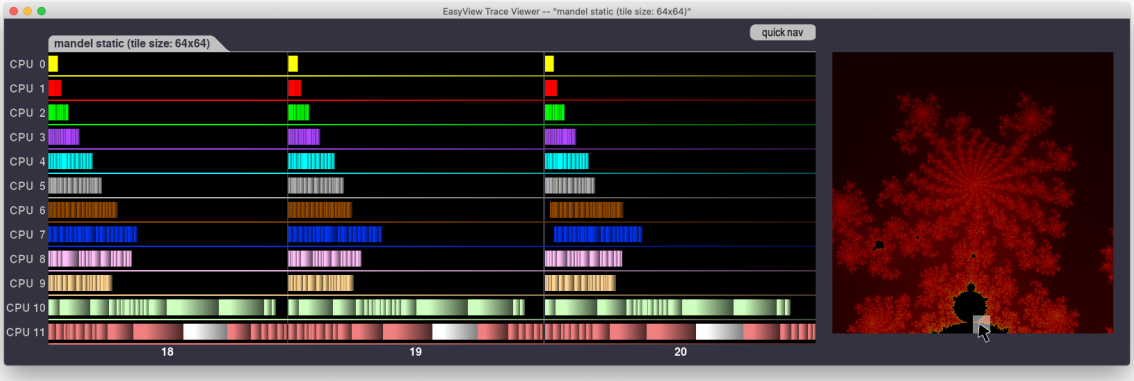


Figure 14: Moving the mouse on the rightmost thumbnail reveals the tasks (highlighted in white color) that have worked on the tile under the mouse pointer.

As an example, we consider the execution of `mandel` using a static distribution of tiles. By executing the application with the trace flag set, we obtain a trace file that we can analyze using EASYVIEW. Figure 14 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 them to check what tiles of the image were the most demanding, for instance. In the situation spotted in Figure 14, 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.

More interestingly, EASYVIEW increases the students' level of understanding of scheduling policies by enabling pairwise comparisons between execution traces. Without such a tool, determining why a

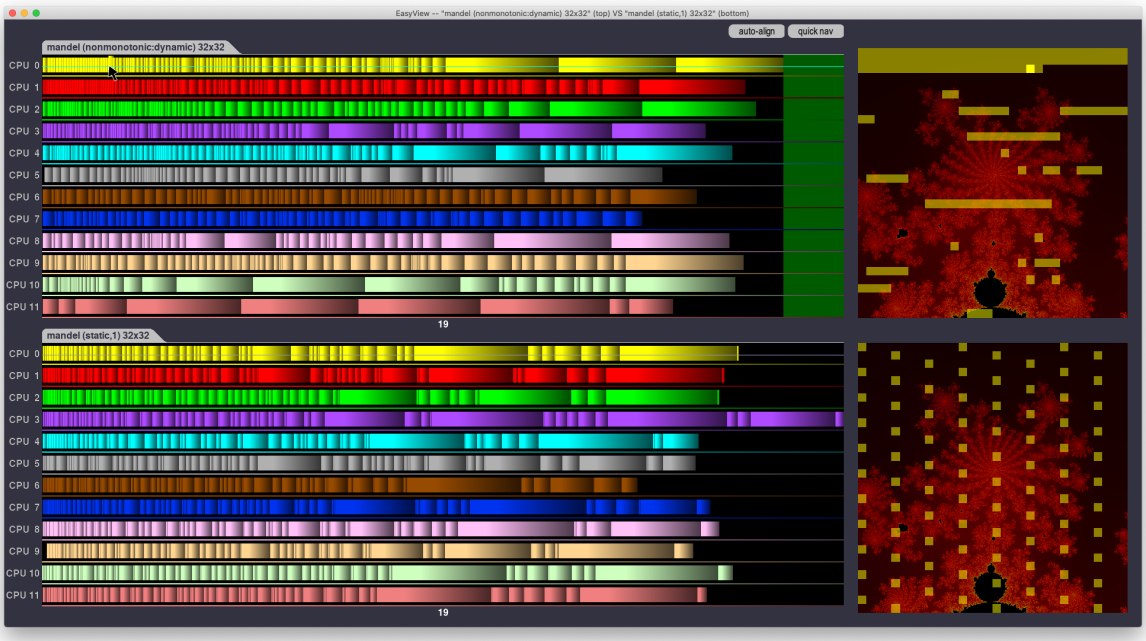


Figure 15: Two executions traces are compared using EASYVIEW. The traces are "re-aligned" so that each iteration virtually begins at the same time on both sides. The green zone indicates the amount of time saved by the fastest execution.

cyclic distribution of work is not performing as well as a dynamic one would be difficult. This is where the *side-by-side comparison* feature of EASYVIEW is useful: the Gantt charts of both scheduling strategies can be aligned iteration per iteration and compared. Figure 15 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). One can observe that the latter strategy does a better (although not perfect) job of 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 using the *coverage mode* feature of EASYVIEW (see Section 3.3) which displays the set of tiles processed by CPU 0 during the observed iterations. Students can immediately recognize the cyclic assignment of tiles to CPU 0 in the bottom thumbnail, and the top thumbnail reveals that CPU 0 has stolen many tiles after completing its initial bunch of tiles under the nonmonotonic:dynamic policy.

5. Example activities

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

5.1. Practical work on multicore programming

After a first hands-on session during which 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 additional data structures. We detail two of these assignments in the remainder of this section.

5.1.1. 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 goal of this practice work is to make students progressively explore the benefits of tiling with respect to cache reuse, measure the influence of code simplification on compiler auto-vectorization, and implement simple heuristics such as the " bigger tasks first" scheduling policy.

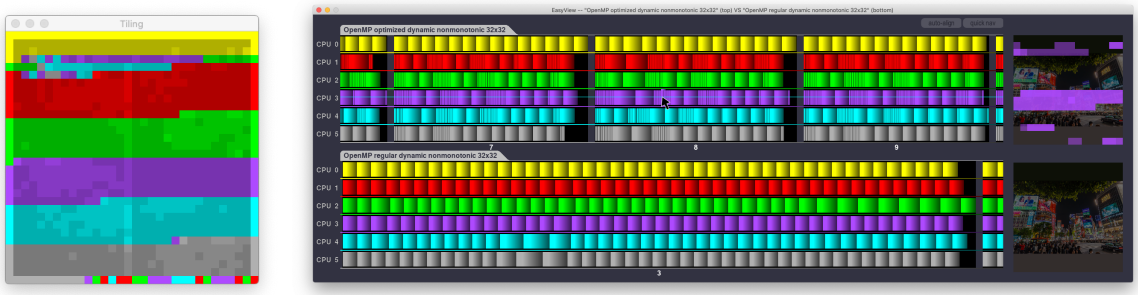
```
unsigned blur_compute_seq (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    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 16: Sequential version of the `blur` kernel

The sequential version of the `blur` kernel (Fig. 16) 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. 16, line 9). 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 neighbors), 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.



(a) Tiling window in Heat map mode

(b) Execution trace comparison

Figure 17: The selective optimization of tiles can be observed by means of two different tools. The heat map mode of the tiling window (Fig. 17a) shows that border tiles take a longer time to be processed than inner tiles. EASYVIEW (Fig. 17b) shows the comparison of two traces. 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).

After implementing these optimizations and parallelizing their code, students can check the correctness of their variant by dumping the final output on disk (in the form of a PNG file) and comparing it with the output of the reference, sequential variant. Once the parallel implementation is valid, students can measure effectiveness by several means. First, they can run EASYPAP in performance mode to realize that the achieved gain is beyond expectations: the new variant is 3 times faster! Then they can activate the *heat map* mode (Figure 17a) to figure out if all tiles are equally cpu-demanding. The tiling window clearly reveals that inner tiles involve less computations than tiles located on the edges. The small computation load observed for inner tiles suggests that the corresponding code has been aggressively optimized by the compiler. To further investigate this observation, EASYVIEW offers a useful

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

trace comparison feature, as shown in Fig. 17b. 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 $10\times$ speedup not only comes from the removal of conditional branches: it is mostly imputable to compiler auto-vectorization ($8\times$ 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. 17b, 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.

For students who want to investigate further, we propose generating 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.

5.1.2. 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 experiment with algorithms where the execution of some tasks has to be delayed until some other tasks have completed.

The target application is 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 neighbors 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 in 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) neighbors have been completed. With OpenMP tasks, these constraints directly translate into task dependencies, as sketched in Fig. 18.

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. 19. Students can observe the order in which tiles were processed by just moving the mouse.

5.2. GPU programming activities

GPU programming using close-to-hardware interfaces (OpenCL, CUDA) is known to be difficult and error-prone. It is nonetheless a mandatory pathway to become familiar with the execution model of GPU accelerators. Fortunately, even low-level technical 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 work-item 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

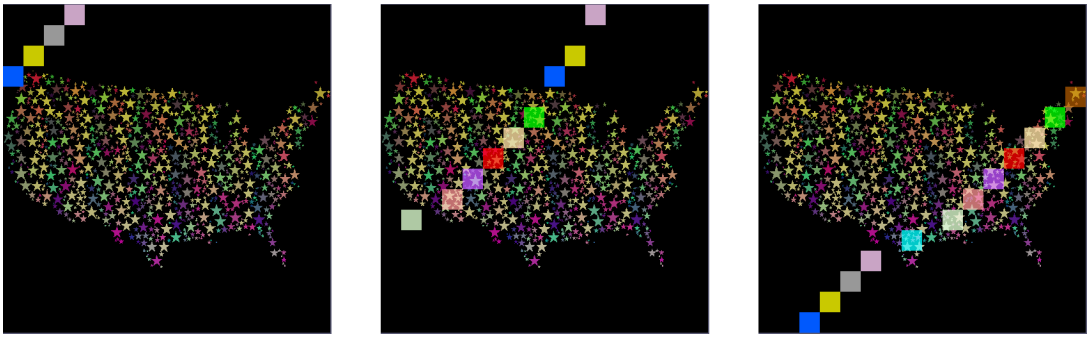


Figure 19: EASYVIEW allows to visualize the wave of tasks moving forward during the execution of code depicted in Fig. 18. These three screenshots were taken while moving the mouse from left to right over the Gantt window.

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.

5.2.1. *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 instruction at each clock cycle, except if some are temporarily inhibited.

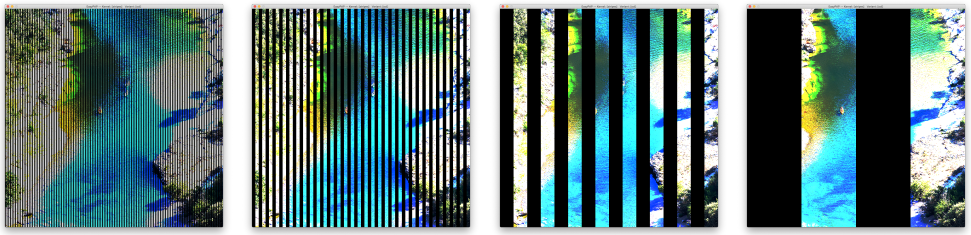
To further explore this constraint graphically, we provide our students with a `stripe` OpenCL kernel that lightens (resp. darkens) vertical stripes of a given image. The code is depicted in Figure 20. The kernel is executed with `DIM` $\times$ `DIM` work-items grouped in horizontal 1024×1 workgroups.

```
__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 20: 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 21), 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 21: Result produced by the `stripe` kernel on a 1024×1024 image for different values of the `arg` parameter.

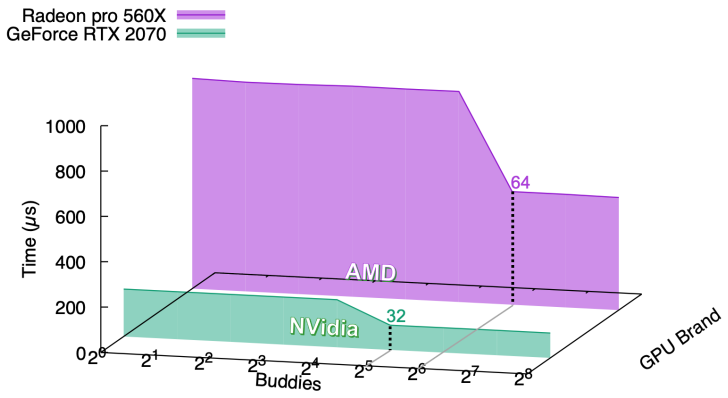


Figure 22: Execution time of the `stripe` kernel for different values of `arg`, on two different GPU architectures. In both cases, there is a threshold after which thread divergence is no longer harmful.

Since the code introduces a divergence (highlighted line in Fig. 20) between work-items 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 in Figure 22. In the end, they observe that a NVidia *warp* is formed by 32 threads, whereas an AMD wavefront is composed of 64 threads.

5.3. Understanding memory coalescing

Coalesced memory transactions represent an important hardware optimization technique used in GPUs to maximize the throughput of load/store instructions. It heavily relies on the fact that threads belonging to the same hardware-defined execution unit (i.e. *half warps* on NVidia hardware) access contiguous memory addresses in global memory.

Many naive implementations of simple image transformation kernels, such as image transposition or image 90-degrees rotation, do not fully benefit from coalesced transactions because half of memory accesses are performed on columns of the image (and are thus not contiguous). The goal of this practice work is to illustrate how local memory can be used as a smart cache so as to perform only contiguous global memory accesses. Students are provided with a naive variant of a 90-degrees rotation OpenCL kernel and they have to apply a tile-based caching approach to end up with a better performing variant. When implementing such an optimization, one common mistake is to forget a synchronization barrier between global-to-local and local-to-global memory transfers, leading to the loss of some pixels along the way. By visualizing the resulting image after a few iterations, students observe that some, but not all, pixels have a random value, which helps them to realize that the problem comes from the missing barrier. In the end, students observe a 20% decrease of the overall kernel duration, which confirms the relevance of their optimization.

6. The capstone assignment

During the second half of the Parallel Programming course, our graduate students work on a capstone assignment which consists in progressively parallelizing a given application using various parallel programming paradigms. As an example, we present last year's assignment which consisted in studying the advanced parallelization of a 2D stencil code, such as Conway's *Game of Life* [14] or Bak-Tang-Wiesenfeld' *Abelian Sandpile Model* [5, 9]. Stencil codes are good candidates for such a "*putting it all together*" activity because they offer numerous optimization opportunities, they can be easily ported on a GPU or distributed over a cluster of machines.

We first present how the project is decomposed into intermediate assignments. Then we present two examples of advanced assignments to illustrate what students do typically achieve. We conclude by presenting the tools and guidelines we give to students to encourage them to adopt a scientific approach.

6.1. *Assignment organization*

The capstone project is carried out by pairs of students, and is divided into four assignments. For each assignment, students have to provide their source code, as well as a scientific report highlighting their approach.

The first assignment is centered around the basic OpenMP parallelization of the 2D stencil. Students are expected to implement some simple optimizations to avoid false sharing and unnecessary synchronizations. They are also asked to experimentally determine the most suitable scheduling policy. They have to explain and justify their approach with the help of performance plots and traces. In return, we give them formative feedback through annotation of their reports and individual or group discussions. A list of recommendations is sent to all students. Actually, this first assignment is a warm-up exercise and is not graded.

In the context of the second assignment, students work on the optimization of memory bandwidth and the introduction of algorithmic optimization by avoiding recalculating stable areas of the domain. They have to mitigate the load imbalance introduced by this latter optimization with "sparse" configurations. Students are asked to report on their progress by extending (and enhancing) their previous report. Again, we give them formative feedback in return. Grading is facilitated by the automatic evaluation of their code: we use scripts to check the correctness of each variant, and we establish a global ranking of the solutions using the obtained speed up as a score. The third assignment is organized the same way. The topic focuses on SIMD vectorization and GPU programming.

The fourth and last assignment is done individually. Students choose the topic that they want to address among a list of proposed assignments. For instance, they can perform an in-depth investigation of a specific topic (e.g., cache oblivious algorithms), or they can combine different programming paradigms to address hybrid or heterogeneous architectures (e.g., MPI + OpenMP, MPI + OpenCL, OpenMP + OpenCL).

6.2. *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.

We use Conway's *Game of Life* [14] for such an advanced assignment. For the sake of simplicity, no load balancing between the CPUs and the GPU is planned in the first phase: the frame is split 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 transferred from host memory to the GPU, and its sibling must be transferred 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 23). They can also observe the cost of data transfers.

Figure 23: 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).

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.

6.3. 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, we use the *Game of Life* example again. This time, we ask students to develop a more elaborate "lazy" implementation that avoids recomputing tiles whose neighborhood 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: neighbor processes must not only exchange the cells themselves, but they also have to exchange meta-information regarding the state (steady or lively) of the tiles containing these cells. 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 3.5. 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
```

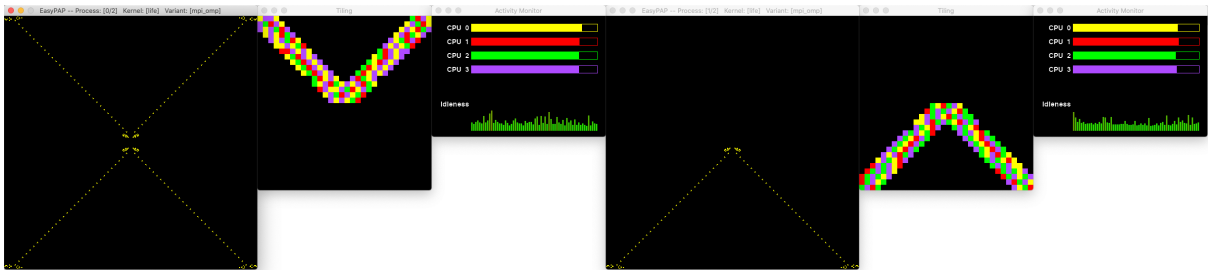


Figure 24: When launched in debugging mode, monitoring windows of every MPI process show up and help to visualize which area is processed by each of them.

The monitoring windows (Fig. 24) confirm that each process contains 4 threads and works on half of the image. Most importantly, since the sparse dataset consists in *gliders* evolving along the diagonals of the image, we can check that only tiles located near diagonals are computed.

6.4. Encouraging a scientific experimental approach

As mentioned previously (Sec. 6.1), guiding students toward the writing of a scientific report is a priority for us. Admittedly, conducting and justifying a campaign of experiments is a difficult task, especially because most of our students have never conducted such a campaign previously. Moreover, the behavior of a parallel program on a multicore machine is difficult to understand, even for a specialist. Indeed, parallel machines are now complex systems that make program execution very sensitive to many parameters (e.g., tile shape or scheduling strategy). We thus strive to teach students to conduct many experiments to compare their implementations and to understand the impact of each parameter, just as we do as researchers.

For this purpose, we have integrated several features into EASYPAP to facilitate the collection of experimental data and their exploitation through the generation of graphs. Each time EASYPAP is invoked in performance mode (i.e., with the `--no-display` flag), it reports the completion time as well as all execution and configuration parameters in a *Comma Separated Value* (`CSV`) file. The collected data can then be exploited by EASYPLOT, a plotting facility based on the `Seaborn` [31] data visualization library and the `pandas` [26] data analysis toolkit. Students can easily filter and select appropriate data from the performance file. Other options are available to specify figure-level or axes-level parameters. 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 plot-lines are set

using the remaining parameters (see Fig. 26b). This guarantees that experiments conducted in different conditions will not silently be incorporated in the same graph.

Most importantly, we provide students with a tutorial explaining how to explore the space of parameters using plots and how to leverage trace analysis to better describe the parallel behavior of their solution. We start by describing how to produce a data set covering the spectrum of parameters to be investigated, using simple python scripts. Then we illustrate how to explore this data set using graphs in order to select the most interesting experiments. At this stage, we detail how to extend their preliminary script so as to generate a statistically sound data set and to produce scientifically usable graphs. Based on the obtained graphs, we formulate several hypotheses and, for each of them, we show how it can be confirmed (or rejected) either by analysing execution traces or by checking hardware characteristics (e.g., L3 cache size). The last part of the tutorial is devoted to guidelines regarding the writing a scientific document presenting the experimental approach and the obtained results.

```
from expTools import *

easypapOptions = {
    "--kernel ": ["mandel"], "--variant ": ["omp_tiled"], "--iterations ": [20],
    "--size ": [1024,2048],
    "--tile-height ":[2**i for i in range(0, 5)],
    "--tile-width ":[2**i for i in range(3,11)]
}

ompICV = {    # OMP Internal Control Variable
    "OMP_SCHEDULE=": ["dynamic","static,1"],
    "OMP_NUM_THREADS=": [46], "OMP_PLACES=":["cores"]
}

execute('./easypap', ompICV, easypapOptions, nbrun=5)
```

Figure 25: Experiments automation script to investigate the role of tile size.

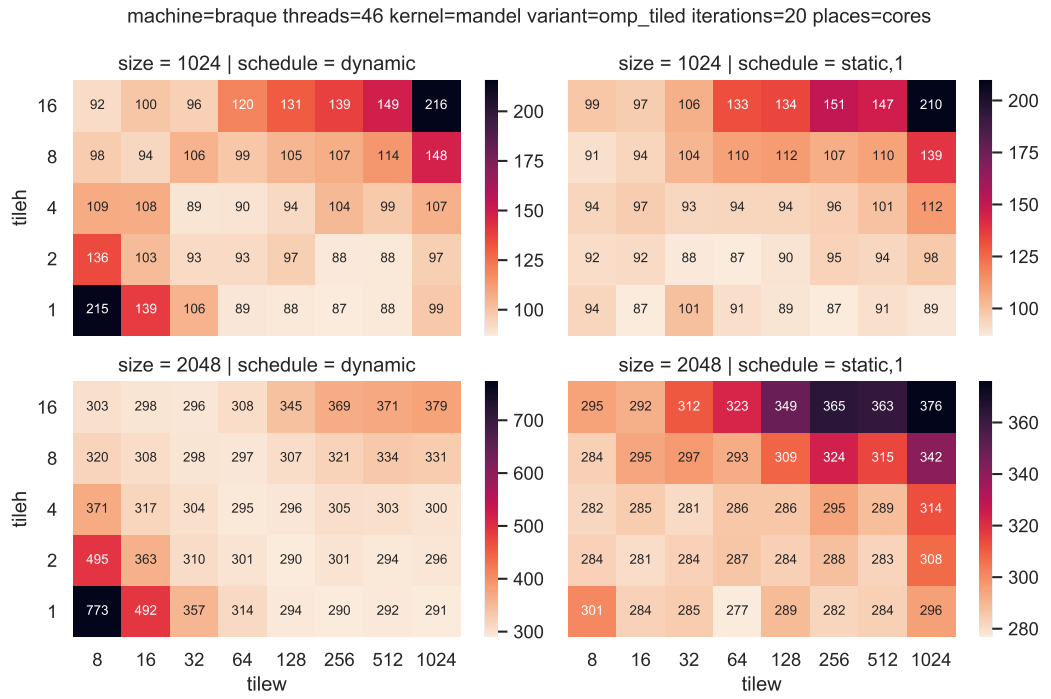
As an illustration, Figure 25 presents a simple script which automates the execution of the OpenMP tiled variant of the Mandelbrot kernel for several image sizes, tile sizes, and scheduling policies. This script has been launched on 48-core machine equipped with $2\times$ Xeon Gold 5118 processors. Figure 26a shows the heat map obtained with EASYPLOT, which reprensents a concise way of observing the influence of the tile geometry on the overall performance, using two distinct scheduling policies and two different image sizes. This heat map impelled us to perform additional experiments, using the best geometry for each case. Figure 26b reports the evolution of performance with respect to the number of threads involved. Interestingly, performance drops when 48 threads are used. In our tutorial, we discuss and explain this surprising phenomenon by using traces. In this case, it appears that the cyclic scheduling policy leads to a pathological workload distribution[4]. We feel it is important to share and discuss such imperfect real-life plots with students.

As a final note, we emphasize that the accuracy of titles and legends in students reports (see for instance Fig. 26a) is probably one of the greatest improvement brought by the EASYPLOT plotting facility.

7. Pedagogical considerations

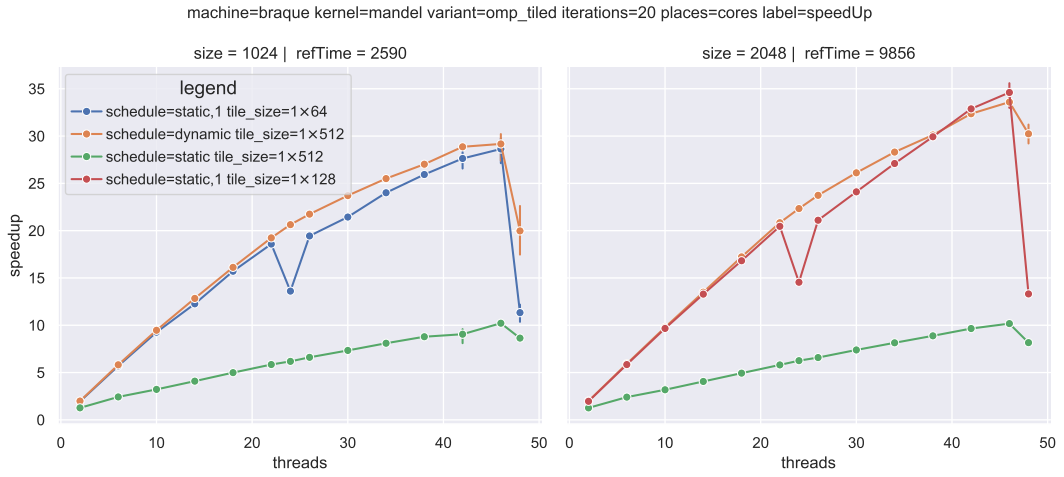
In this section, we report on our teaching experience in the context of the Parallel Architecture Programming Course. First we use Bloom's taxonomy [6] to classify our pedagogical activities in order to demonstrate the usefulness of EASYPAP at all levels of learning. Then we present students' responses to an anonymous survey regarding the relevance of EASYPAP as a software and as a pedagogical tool. Finally, we comment on the evolution of our Parallel Programming course and we discuss on the lessons learned since we began using EASYPAP.

[4]Indeed, we have $\frac{2048}{128} = \frac{1024}{64} = 16 = \frac{48}{3}$ therefore the image is split into 16 columns, each beeing processed by 3 threads. Unfortunately, it turns out that the middle column takes much longer to compute than the others.



(a) Heat map revealing the influence of tile geometry on performance. Numbers inside tiles denote the average duration time (in ms). This heat map was obtained using the following command:

```
easyplot --kernel mandel --plottype heatmap -heatx tilew -heaty tileh --col schedule --row size
```



(b) Speedup plots of `mandel` kernel. The speedup ratios were computed against the best sequential execution time (2590 ms for 1024 × 1024 and 9856 ms for 2048 × 2048). This graph was obtained using the following command:

```
easyplot --kernel mandel --label speedUp --col size
```

Figure 26: Some plots given in the experiment tutorial and their associated command line.

Cognitive Process	Student's engagement activities with the help of EASYPAP
Remember *Retrieving relevant knowledge from long-term memory*	As mentioned in Section 4.1, using EASYPAP involves students' 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 4.3.
Understand *Determining the meaning of instructional messages. Including oral and graphic communication.*	During lectures and labs, we use EASYPAP with different codes to illustrate various parallel concepts (Sec. 4.1) and to observe the concrete meaning of keywords (Sec. 4.2).
Apply *Carrying out or using a procedure in a given situation.*	As mentioned in Section 5, 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 they use the same environment for many lab sessions and they can visually check their progress.
Analyze *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) establish links between an issue and the source code and (3) modify the code to improve it. Examples of such optimizations are presented during lectures.
Evaluate *Making judgment based on criteria and standards.*	Students have to write a report in the context of a capstone assignment (Sec. 6). In this report they are asked to follow a scientific approach based on performance graphs and trace analyses: (1) select data to produce meaningful performance graphs, (2) explain the role of main parameters. Students are provided with a tutorial presented in Section 6.4 to guide them through this activity.
Create *Putting elements together to form a novel, coherent whole or make an original product.*	Students have to design new parallel algorithms (and data structures) and to write a scientific report in the context of the capstone assignment.

Table 2: Classification of some educational activities with EASYPAP according to Bloom's taxonomy.

7.1. Bloom's levels of understanding

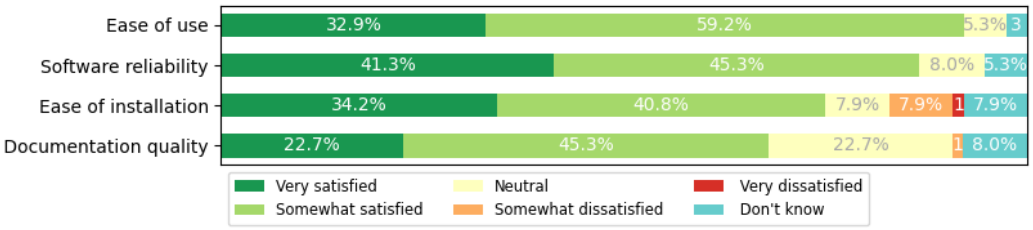
We believe that the EASYPAP environment is a valuable tool to engage learners, not only because it offers an attractive graphical experience, but also because it enables students to acquire new knowledge at different *levels of understanding*. By levels of understanding, we refer to the famous taxonomy of Bloom [6], which is a way to categorize the levels of reasoning skills required in classroom situations. In Table 2, we classify some of our activities with EASYPAP according to Bloom's taxonomy. We can note that lectures and labs mainly focus on the first four levels, whereas the levels *Evaluate* and *Create* are specifically worked on during the capstone assignment described in Section 6.

7.2. Students' feedback

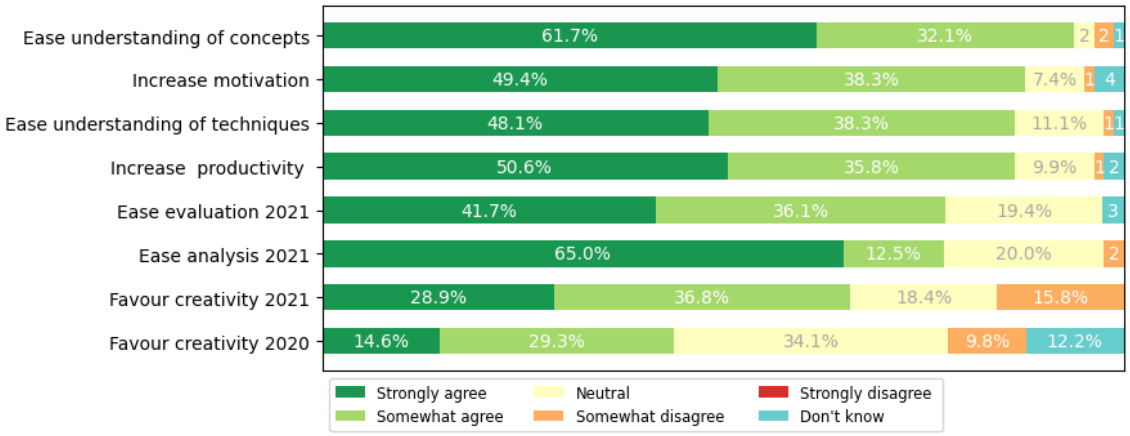
In 2020 and 2021, we conducted an anonymous online survey about EASYPAP. The first round took place in March 2020, just prior to France's Covid'19 lockdown measures. The surveyed population consisted of 59 University computer science students (postgraduate level) and 28 engineering school students. All students had a record of about eight hours of practice with EASYPAP. The capstone assignment was given during the lockdown period, so the survey did not cover this homework activity. The second round took place in March 2021 and involved 61 master students. In contrast with the previous year, it was conducted after the students had already accomplished half of their long-term project (see Sec. 6). The purpose of the survey was clearly indicated to students: the goal was to collect objective feedback about EASYPAP and publish the results in a scientific article. In total, we gathered answers from $27 + 41$ Master students and 14 engineering students, which represents a response rate of 55%. 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.

A summary of the answers regarding the EASYPAP software is presented in Figure 27a. One can observe that students are mostly satisfied with the software. However, some students have experienced difficulties in installing EASYPAP at home during the lockdown period. Most issues were related to finding an appropriate OpenCL driver compatible with their custom installation, which is admittedly often challenging. Nevertheless, after a brief period of panic, all students were able to use either their personal computer or University's servers remotely to perform their experiments. To this end, we added several features to EASYPAP to ease its utilization through a remote connection.



(a) Software quality of EASYPAP



(b) Impact of EASYPAP on pedagogy

Figure 27: Summary of survey responses about EASYPAP.

The goal of second part of the survey was to figure out how students would describe the pedagogical impact of the EASYPAP approach. To this end, each question was directly related to a specific level of Bloom's taxonomy. Regarding the *comprehension* level, we wanted to distinguish the understanding of theoretical concepts (e.g., load balancing, need for synchronization mechanisms) from more technical concepts (e.g., parallel architectures, parallel programming environments). For the *apply* level, we have also asked two questions: one about motivation (stimulation during labs) and another one about productivity (student's efficiency). Questions about *analysis* and *evaluation* levels were introduced in 2021. Regarding the analysis level, we asked students whether the use of traces and graphs helps to identify and address performance problems by facilitating the link between the identified behavior and the code. For the evaluate level, we asked whether the environment for producing traces and graphs has helped them to follow a scientific approach and to write a better report. Finally, for the *creativity* level we simply asked them if using EASYPAP gave them new ideas about code optimizations or about other contexts in which they could exploit their skills.

Answers to this second part of the survey are presented in Figure 27b. About 86% of the students acknowledge that EASYPAP has helped them to better understand parallelism and has improved their efficiency. The satisfaction rates on the levels of analysis and evaluation are higher than 77%. The impact of using EASYPAP is rather positive: the analysis of parallel program performances is an activity requiring a fair expertise that we want to share with students. The results about creativity are more mixed, however we can see the influence of the project on the creativity level: the satisfaction rate on this purpose has increased from 44% in 2020 to 65% in 2021.

The last part of the survey was devoted to open-ended comments. We received 17 comments in

which the students discussed the aspects of EASYPAP they appreciate the most. Most of all emphasize the motivation brought by EASYPAP and the fact that they were able to mostly focus on parallelism. We have also received some relevant suggestions. 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.

7.3. Discussion

Since we introduced EASYPAP in our lectures and lab sessions, students clearly find parallel programming much more attractive and fun. During lab sessions, EASYPAP's graphical facilities made debugging phases less painful and more effective. Students quickly prototype preliminary variants of their code and analyze their parallel behavior interactively. The fact that they only need a few days to achieve an efficient MPI+OpenMP implementation of the *Game of Life* kernel using lazy evaluation (see Section 6.3) is also a tangible improvement. Moreover, EASYPAP provides a unified environment for programming, experimenting, data mining and analyzing parallel kernels. Students spend more time investigating new ideas and conducting thorough experiments. We also believe that student reports are much more rigorous, better argued and more interesting than before.

EASYPAP also proved to be a great companion during parallel programming lectures, by injecting dynamism in our lectures and allowing us to progressively explore new concepts using an interactive *test-fail-retry* approach. Our experience is that it is sufficient to use EASYPAP during a few lessons only, either to introduce a new topic or to observe the reaction of students to surprising parallel behaviors. We must also admit that we have learned a lot on our side, and it turns out that we had the opportunity to observe behaviors that would normally require the use of complex profiling tools. For instance, by comparing the execution traces generated with different compilers (namely gcc and clang), we have been able to visually observe and compare the different task scheduling strategies used by the underlying OpenMP runtime systems.

Obviously, using EASYPAP also has several downsides. The first one lies in the fact that we provide 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 practice works as well, involving writing small applications from scratch to show students what a complete parallel program looks like. The second one is a consequence of placing excessive confidence in the graphical output of programs. In 2020, we were surprised to find that half of the students had submitted at least one buggy implementation even though they had all the tools to check its correctness. The reason lies in the fact that the resulting images displayed on the screen "looked correct", but the faulty areas where not visible to the human eye. In 2021, the situation has improved a little (the ratio of buggy implementations dropped to a quarter) because we have explicitly drawn students' attention on this problem. For the coming years, we plan to implement an automatic grading system to save time and to improve the quality of the submitted assignments, as was successfully deployed at Rice University [17].

8. Related Work

The design of the educational EASYPAP parallel programming framework is a work at the crossroad of research efforts about interactive visualization, profiling and off-line analysis of parallel programs, and educational programming environments.

8.1. Interactive visualization of parallel programs

Regarding interactive visualization, we adhere to the same philosophy as ParaVis [8] and TSGL [2], which provide easy-to-use C/C++ interfaces to visualize 2D results produced by parallel computations. TSGL (Thread safe graphics library) is a C++ library that was designed to allow multiple threads (Pthreads or OpenMP) to draw predefined shapes (polygons, lines, text, images) on a shared canvas. The color of a shape reflects the ID of the thread by which it was drawn, so as to easily visualize the contribution of each thread on the screen. Paravis is another C++ library aiming at improving bug detection in parallel programs running on multicore machines as well as NVidia GPU accelerators. The application is responsible for filling a preallocated matrix of pixels and for updating the graphical window when appropriate.

TSGL and Paravis have both influenced the design of EASYPAP, which also provides 2D interactive visualization facilities. However, EASYPAP uses a separate *tiling window* to identify the contributions of each processing unit, rather than altering the pixels of the main window. In contrast with TSGL and

ParaVis, which are libraries that can be integrated into almost any existing C/C++ application, EASYPAP follows a different approach by providing an integrated educational framework with monitoring and trace exploration capabilities, experience automation and plot generation assistance.

8.2. Profiling and off-line analysis of parallel programs

Many outstanding tools have been developed to monitor parallel codes and to analyze the obtained performance metrics.

ViTE [33] is a portable and open-source profiling tool designed to visualize execution traces produced by parallel applications. It relies on the semantic-free Paje [20] trace input format in order to support any kind of traces, and it uses an OpenGL canvas with GPU acceleration to improve rendering performance. ViTE has largely inspired the design of our EASYVIEW trace visualizer, and both software provide a similar interactive timeline experience. EASYVIEW's most notable additions are the capability of comparing two traces side-by-side with an auto-alignment feature, and the display of the data region accessed by each task.

Intel Vtune [34] is a powerful profiler and performance analyzer for x86 architectures. Vtune leverages Hardware performance counters and sampling techniques to provide a comprehensive set of tools helping to identify code hotspots, to check CPU utilization, to evaluate vectorization performance, to reveal thread synchronization latencies, to check cache hit ratios, or to highlight memory contention problems. TAU (Tuning and Analysis Utilities) [28] is a comprehensive toolset for the performance instrumentation, analysis, and visualization of large-scale parallel programs. It offers several ways to instrument parallel programs, either automatically via a code instrumentor, dynamically through the Java Virtual Machine runtime, or manually using an instrumentation API. Other tools with similar functionalities include Vampir [18] and Paraver [27]. By making it possible to diagnose performance problems, all these tools offer more powerful profiling capabilities than EASYPAP, but they are also much more difficult to learn for beginners. By providing easy-to-use, built-in profiling tools, we think that EASYPAP represents an attractive stepping stone to these sophisticated tools.

Other environments were designed from the start as *task-oriented* performance analysis tools. Grain Graphs [21] is an OpenMP performance analysis method that visualizes computations performed tasks or parallel for-loop chunk instances, highlighting problems such as low parallelism or poor parallelization benefit at a fine grain level. Aftermath [11] is a tool for interactive off-line visualization, filtering and analysis of execution traces of OPENSTREAM task-based programs. One of its most interesting feature is the ability to set the color of tasks according to the value of selected statistics collected from hardware counters. Aftermath can thus easily turn the timeline into a heatmap highlighting the locality of memory accesses or the branch misprediction rate for each individual task. The StarVZ framework [15] provides visualization panels combining customizable statistics about tasks (number of ready tasks, critical path, etc.) to better cope with the stochastic behavior of task schedulers and pinpoint performance problems incurred by the underlying scheduler.

The visualization facilities offered by these environments were specifically designed for task-based systems and are, in this respect, more powerful than those provided by EASYPAP. EASYPAP supports a wider range of parallel paradigms and is not limited to task-based parallelism. Nonetheless, it provides original tools to help observing the scheduling of tasks of a kernel variant. First, the interactive *task-data mapping* displayed by EASYVIEW allows to check the order in which tiles have been executed. Even if EASYPAP does not collect information about task dependencies, the user can check if task execution order conforms to the expected pattern of task dependencies (as illustrated in Fig. 19). Second, EASYPAP brings a *CPU coverage* functionality to evaluate the amount of data locality exhibited by a given execution trace, which is a very useful metric regarding task scheduling algorithms.

8.3. Teaching Methodologies

In [22], the authors propose a parallel programming teaching methodology based on practical experiences conducted during classes. They demonstrate the importance of teaching students how to analyze performance metrics through the use of appropriate tools. The authors emphasize that teachers should discuss and analyze these metrics with students. Our experience is completely in line with their conclusions, and we think that EASYPAP's profiling, tracing and plotting tools favor interactions between instructor and students.

Like the authors of [1], we are convinced by the pedagogical benefits of using exemplars. EASYPAP was designed around a similar idea: students are provided with operational sequential codes that they can parallelize using multiple paradigms and multiple variants. Our approach hides many technical

details related to initialization of SDL, OpenCL or MPI and provides a simple coding environment where students can really concentrate on parallelism. We have also managed to keep the same kernel interface for multicore, OpenCL and MPI kernels.

During the EduPar 2018-2019 and EduHPC 2018-2020 workshops, twenty-two peachy assignments [35] were presented. These are turnkey activities, easily adoptable for teachers and fun and inspiring for students. We observed that nine assignments are iterative 2D stencils written in C/C++ that could easily be integrated into the EASYPAP kernel database. Moreover, three peachy assignments [4, 3, 7] presented by the TRASGO team of the University of Valladolid attracted our attention because the underlying approach is based on gamification [13, 16]. The authors have designed TABLON, a tool aiming at organizing performance contests and providing statistics needed to implement gamification techniques. We believe that EASYPAP and TABLON are complementary environments.

9. 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 them 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 [30].

References

[1] 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.

[2] 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.

[3] M. Agung, M. A. Amrizal, S. Bogaerts, R. Egawa, D. A. Ellsworth, J. Fernandez-Fabeiro, A. Gonzalez-Escribano, S. Kundu, A. Lazar, A. Malony, H. Takizawa, and D. P. Bunde, "Peachy parallel assignments (eduhpc 2019)," in *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, 2019, pp. 75–83.

[4] E. Ayguadé, L. Alvarez, F. Banchelli, M. Burtscher, A. González-Escribano, J. Gutierrez, D. A. Joiner, D. R. Kaeli, F. Previlon, E. Rodriguez-Gutiez, and D. P. Bunde, "Peachy parallel assignments (eduhpc 2018)," in *2018 IEEE/ACM Workshop on Education for High-Performance Computing, EduHPC, Dallas, TX, USA, November 12, 2018*. IEEE, 2018, pp. 78–85. [Online]. Available: https://doi.org/10.1109/EduHPC.2018.00012

[5] P. Bak, C. Tang, and K. Wiesenfeld, "Self-organized criticality," *Physical Review A*, vol. 38, pp. 364–374, 07 1988.

[6] B. Bloom, M. Engelhart, E. Furst, W. Hill, and D. Krathwohl, *Taxonomy of educational objectives : the classification of educational goals. Handbook I, Cognitive domain / by a committee of college and university examiners ; Benjamin S. Bloom, Editor*. David McKay New York, 1956.

[7] H. Casanova, R. da Silva, A. Gonzalez-Escribano, W. Koch, Y. Torres, and D. P. Bunde, "Peachy parallel assignments (eduhpc 2020)," in *2020 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 53–58. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/EduHPC51895.2020.00012

[8] A. Danner, T. Newhall, and K. C. Webb, "Paravis: A library for visualizing and debugging parallel applications," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 326–333.

[9] D. Dhar, "Self-organized critical state of sandpile automaton models," *Phys. Rev. Lett.*, vol. 64, pp. 1613–1616, Apr 1990. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.64.1613

[10] A. Douady and J. Hubbard, "Exploring the mandelbrot set. the orsay notes." 10 2009, (Visited on 2021-30-03). [Online]. Available: http://pi.math.cornell.edu/~hubbard/OrsayEnglish.pdf

[11] 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.

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

[13] J. Fresno Bausela, H. Ortega-Arranz, A. Ortega-Arranz, A. Gonzalez-Escribano, and D. Ferraris, "Applying gamification in a parallel programming course," in *Gamification-Based E-Learning Strategies for Computer Programming Education*. IGI Global, 01 2017, pp. 106–130.

[14] M. Games, "The fantastic combinations of john conway's new solitaire game "life" by martin gardner," *Scientific American*, vol. 223, pp. 120–123, 1970.
[15] V. Garcia Pinto, L. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task-based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, vol. 30, p. e4472, 04 2018.
[16] 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.
[17] 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.
[18] 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.
[19] M. A. Kuhail, S. Cook, J. W. Neustrom, and P. Rao, "Teaching parallel programming with active learning," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 05 2018, pp. 369–376.
[20] S. L. Mello, F. Mathieu, T. François, O. S. Benhur, K. J. C. T. P. trace file format. : UFRGS; 2016.B. de Oliveira Stein, and J. C. de Kergommeaux, "The paje trace file format," UFRGS, Tech. Rep., 2016.
[21] 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.
[22] R. Muresano Caceres, D. Rexachs, and E. Luque, "Learning parallel programming: a challenge for university students," *Procedia CS*, vol. 1, pp. 875–883, 05 2010.
[23] R. Namyst and P.-A. Wacrenier. (2018) The EASYPAP web site. (Visited on 2021-30-03). [Online]. Available: https://gforgeron.gitlab.io/easypap/
[24] OpenACC Standard, "The OpenACC Application Programming Interface Version 3.0," Nov. 2019. [Online]. Available: https://www.openacc.org/specification
[25] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," Jul. 2013. [Online]. Available: https://www.openmp.org/specifications/
[26] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134
[27] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, vol. 44, 03 1995.
[28] 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.
[29] 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.
[30] 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.
[31] M. Waskom and the seaborn development team, "mwaskom/seaborn," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.592845
[32] "SDL: Simple directmedia layer," (Visited on 2020-03-09). [Online]. Available: https://www.libsdl.org
[33] "ViTE: Visual trace explorer," (Visited on 2020-03-09). [Online]. Available: http://vite.gforge.inria.fr
[34] "Intel Vtune Profiler," (Visited on 2020-03-09). [Online]. Available: https://software.intel.com/en-us/vtune
[35] "Peachy assignments - nsf/ieee-tcpp curriculum initiative on parallel and distributed computing," (Visited on 2021-30-06). [Online]. Available: https://tcpp.cs.gsu.edu/curriculum/?q=peachy