# EASYPAP: a Framework for Learning Parallel Programming

Alice Lasserre, Raymond Namyst, Pierre-André Wacrenier

# EASYPAP: a Framework for Learning Parallel Programming

Alice Lasserre* and Raymond Namyst† and Pierre-André Wacrenier†
*Computer Science department, University of Bordeaux*
Talence, France
*alice.lasserre@etu.u-bordeaux.fr, †{raymond.namyst, pierre-andre.wacrenier}@u-bordeaux.fr

***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**

## I. INTRODUCTION

During the last decade, the High Performance Computing community had a hard time coping with the evolution of parallel architectures toward massively parallel heterogeneous multicore machines. In fact, all software developers are currently concerned about this manycore trend which has impacted every commodity hardware, from smartphones to desktop machines. To get the most out of nowadays computers, people must be trained to parallel programming. It is no surprise that integrating HPC into undergraduate and postgraduate courses to expose all students to basic parallel programming skills has been identified as a priority by the *European Technology Platform for HPC* in their 3rd Strategic Research Agenda [8].

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, intrinsics instructions, or a mix of them. EASYPAP was designed to make it easy to implement multiple variants of a given kernel, and to experiment with and understand the influence of many parameters related to the scheduling policy or the data decomposition. During our undergraduate and postgraduate lab sessions, students enjoyed the feedback provided by the graphical tools and were able to gain a deeper understanding of both parallel programming and computer architecture.

## II. THE EASYPAP FRAMEWORK

EASYPAP is a C programming environment that relies on the SDL library [2] to interactively render the results of 2D computations. EASYPAP's main philosophy is to let students focus on computation kernels while hiding most of the implementation details related to program initialization (at the notable exception of memory allocation), code instrumentation and interactive display. The `main` program loop is thus controlled by EASYPAP.

### A. 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 obviously be easily added. Let us take the `mandel` kernel, which computes the Mandelbrot set, as an illustration. Fig. 1

```
1 void mandel_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     zoom (); // modify the viewpoint real coordinates
8   }
9 }
```

Figure 1. Sequential version of kernel mandel

shows a straightforward sequential implementation of the `mandel` kernel.

The outer loop (line 3) performs the requested `nb_iter` iterations in a row. Lines 4–6 illustrate how the contents of the image are accessed during an iteration. For the sake of simplicity, EASYPAP works on square shape images. The pixels of the image are accessed through the `cur_img(row, col)` macro. Here is how to run the `seq` variant of the `mandel` kernel on a $2048 \times 2048$ image:

```
easypap --kernel mandel --variant seq \
        --size 2048
```

This action brings a window on the screen which displays an animation consisting of the series of images computed at each iteration. The animation can be paused, or can be slightly accelerated by skipping frames.

Since checking the belonging to the Mandelbrot set may be computed independently for any $(i, j)$ pixel, the `mandel` 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 `mandel_compute_omp`, insert a single "`#pragma omp parallel for`" clause before the `for` loop iterating over lines, recompile EASYPAP, and launch:

```
easypap --kernel mandel --variant omp
```

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

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

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

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

### B. Interactive monitoring

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

Fig. 2 shows a typical OpenMP tiled implementation of the `mandel` kernel where the `do_tile` function has been instrumented. This function sequentially computes all the pixels inside an arbitrary rectangle defined by `(x,y,width,height)`. 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 has been instrumented, real-time monitoring can simply be activated:

```
easypap --kernel mandel --variant omp_tiled \
        --tile-size 16 --monitoring
```

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

The **Activity Monitor window** reports the real-time load of each CPU. This load is a percentage representing the amount of time spent in computations over the duration of the iteration. At the bottom of the window, a history diagram reports the evolution of *cumulated idleness* over time. In Fig. 3, we clearly observe a load imbalance between

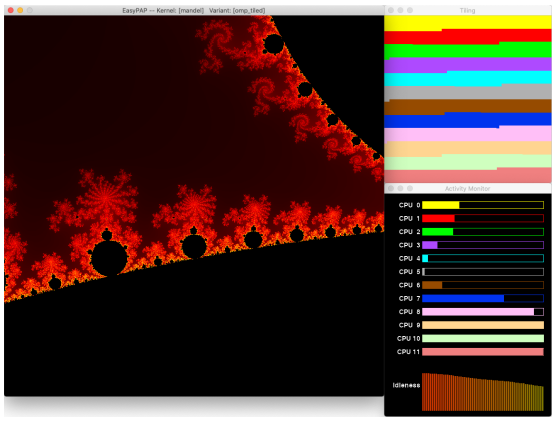


Figure 3. The monitoring mode displays two additionnal windows: a tiling window (top) and a CPU monitoring window.

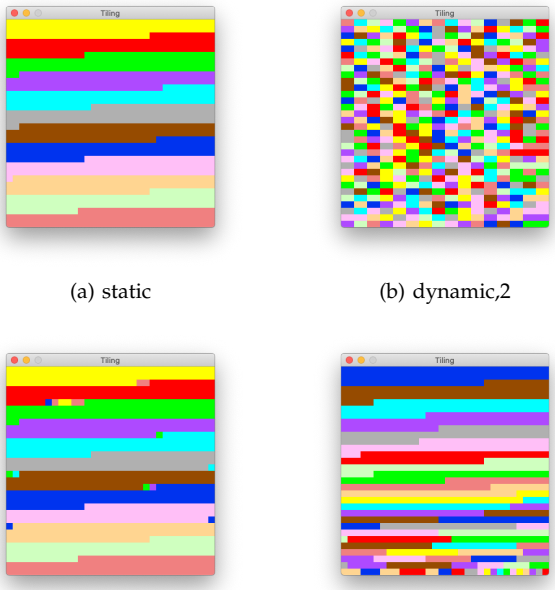
CPUs. The static distribution of tiles is indeed inappropriate because the large black area at the bottom of the image, which contains a lot of pixels belonging to the Mandelbrot set, involves much more computations than other areas.

The **Tiling window** reflects the way tiles have been assigned to threads at each iteration. Each thread is assigned a different color which is consistent with the color assigned to CPUs in the *Activity Monitor* window. By observing Fig. 3, we see that the tiles have been assigned to threads in contiguous blocks, in accordance to the *static* loop scheduling policy.

The tiling window is a precious tool to observe the different loop scheduling policies of OpenMP when combined with the `collapse` clause. In Fig. 4, we examine various loop scheduling policies through the tiling window. Fig. 4a shows that the `static` clause evenly distributes tiles to threads in contiguous chunks. Fig. 4b reveals the opportunistic nature of the `dynamic` clause. Fig. 4c illustrates the behavior of the new OpenMP 5 `nonmonotonic` clause: tiles are first distributed in a static manner, but *work-stealing* is eventually used to correct load imbalance. Finally, Fig. 4d shows how size of chunks assigned to threads decreases over time with the `guided` policy.

### C. Performance mode

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



(a) static (b) dynamic,2

(c) nonmonotonic:dynamic (d) guided

Figure 4. During execution, students observe how the OpenMP loop scheduling policy impacts the assignment of tiles to threads. Note that (a) reflects a steady state, whereas (b), (c) and (d) are dynamically changing between iterations.

wall clock time after completion of the requested number of iterations.

```
> easypap --kernel mandel --variant omp_tiled \
          --tile-size 16 --iterations 50      \
          --no-display
50 iterations completed in 579ms
```

Moreover, the completion time, together with all execution and configuration parameters, are reported in a *Comma Separated Values* (`CSV`) file. Students can customize simple python scripts to automate their experiments by specifying parameter ranges, as illustrated in Fig 5.

Students can then exploit their data and produce the desired graph or histogram thanks to the `easyplot` command. A key feature of `easyplot` is that the legend is automatically generated from the data. Once data have been filtered, constant parameters are put aside, and the names of plotlines are set using the remaining ones (see Fig. 6). This guarantees that experiments conducted in different conditions will not silently be incorporated in the same graph, a common mistake among students's reports.

```
from expTools import *

easypap_options["--kernel "]     = ["mandel"]
easypap_options["--iterations "] = [10]
easypap_options["--variant "]    = ["omp_tiled"]
easypap_options["--grain "]      = [16, 32]

omp_icv["OMP_NUM_THREADS="] = list(range(2, 13, 2))
omp_icv["OMP_SCHEDULE="]    = ["static", "guided",
                "dynamic,2", "nonmonotonic:dynamic"]

execute('easypap', omp_icv, easypap_options, runs=10)
```

Figure 5. Typical experiments automation script

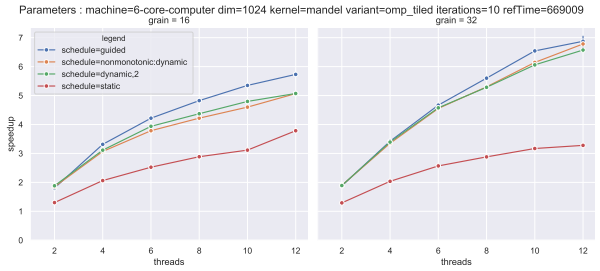



Figure 6. Speedup graphs with 16 × 16 and 32 × 32 tiles. This graph is generated by the command `easyplot --kernel mandel --col grain --speedup` It uses data produced by `easypap` in performance mode. Parameters with unique value are listed above the graph.

### D. Post mortem trace analysis

Although the monitoring facilities greatly help to detect and understand flaws in the execution of kernels, it cannot always capture some subtle properties such as the heterogeneity of tasks duration or the correct implementation of task dependencies. When a deeper analysis is required, students use the `--trace` option to record tile-related profiling events at execution time (i.e. start/end time, tile coordinates, cpu) into a trace file:

```
easypap --kernel mandel --variant omp --trace \
        --no-display --iterations 10
```

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

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

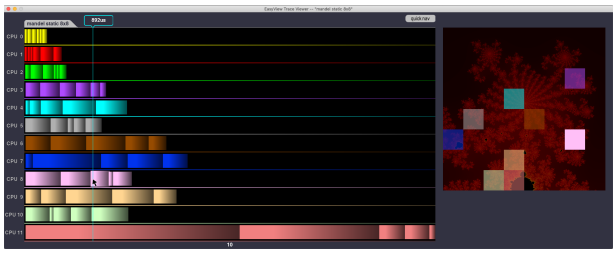


Figure 7. EASYVIEW brings interactive exploration of traces. Moving the mouse over a task in the Gantt diagram displays its duration (bubble at top of window). Tasks intersecting the mouse $x$-axis have their corresponding tile highlighted over the image thumbnail, allowing to link computations and their data.

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

In addition, students can toggle between this vertical mouse mode and a horizontal mode in which the y-axis of the mouse allows to select a particular CPU and highlights the tiles computed during the displayed period. Basically, this allows to observe the "coverage map" of a given CPU during one or multiple iterations, and to check the locality of computations across iterations. This feature is further detailed in Section III-B.

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

## III. EXAMPLE OF ASSIGNMENTS

We have used EASYPAP and EASYVIEW both with undergraduate students during parallel programming introduction courses, and with post-graduate students during parallel and distributed computing courses, where students explore advanced features of multicore, GPU and cluster programming. Even if students usually start with simple, straightforward implementations of basic

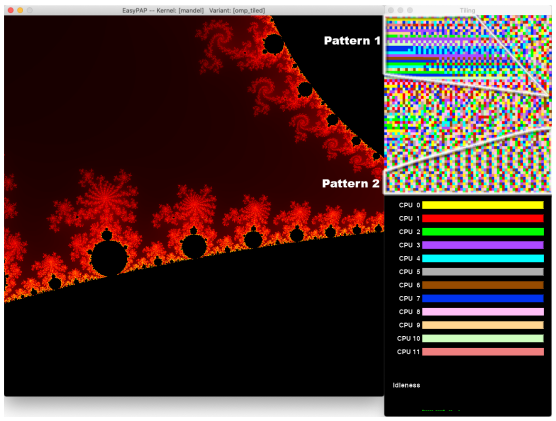
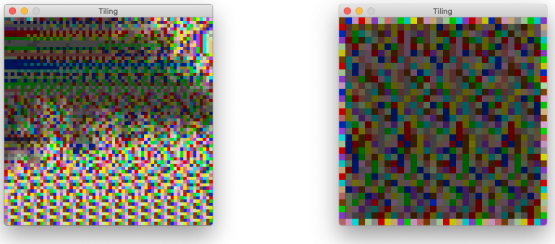



Figure 8. When using OpenMP dynamic loop scheduling of small tiles, the tiling window reveals two noticeable patterns.




(a) mandel (b) blur

Figure 9. In "heat map" mode, the brightness of tiles displayed in the Tiling Window reflects the duration of the corresponding tasks: the brighter an area is, the more time-consuming it is. On picture (a) we can distinguish the shape of the Mandelbrot set as depicted in Fig. 8. On picture (b), we observe that border tiles take a longer time to be processed than inner tiles.

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 very difficult to observe. Animated screenshots of EASYPAP and EASYVIEW are available on the project site [12].

### A. Computing the Mandelbrot Set

After a first hands-on session during which undergraduate students discover the EASYPAP environment using very simple kernels, their first assignment is usually devoted to the computation of the Mandelbrot Set. As we previously mentioned, parallelizing the `mandel` is trivial, but achieving good speedups requires to pay attention to load balancing. The assignment is thus mostly about performing experiments to find the best combination of loop scheduling policy, tile shape and tile size.

Almost all students end up with a tiled implementation, either using Pthreads or OpenMP, similar to the one described in Fig. 2 using dynamic distribution of squared tiles. The size of tiles depends on the dimension of the image as well as on the underlying hardware. After a few iterations, students observe interesting patterns appearing in the *tiling window*, as spotted in Fig. 8.

**Pattern** 1 reveals horizontal stripes of the same color together with a few stripes featuring an alternation of two colors. These stripes correspond to one or two threads computing several tiles in a row. Such a situation happens because 1) these tiles correspond to areas located far away from the Mandelbrot set, where computations take only a few iterations to complete and 2) the other threads are all busy computing time-consuming tiles in the top-right black corner.

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

### B. Picture Blurring: a simple 2D stencil code

During their discovery of parallel computing, our students are quickly exposed to simulations involving *Stencil* computations. We use an assignment based on a *Picture Blurring* kernel to introduce students to the parallelization of 2D stencil codes. The sequential version of the `blur` kernel uses two images. At each iteration, all pixels from the $3 \times 3$ 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.

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

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

*map*" mode of the tiling window: Fig. 9b reveals that inner tiles involve less computations than tiles located on the edges.

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

Another interesting observation can be made when switching to the "*coverage map*" mode provided by EASYVIEW, using mouse horizontal mode to select all displayed tasks for a given CPU. In Fig. 10, 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.

### C. Identification of Connected Components

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

Parallelizing this algorithm without introducing extra iterations is quite challenging. A possible solution is to use a tiled implementation in which tiles are processed with some constraints: during the bottom-right phase (resp. up-left), a tile cannot be executed until its left and upper (resp. right and lower) neighbours have not completed. With OpenMP tasks, these constraints directly translate into task dependencies, as sketched in Fig. 11.

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. 12. One can observe the order in which tiles were processed by just moving the mouse.

### D. Game of Life: Putting it All Together

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 present an assignment to students who attend advanced courses on HPC programming.

The goal is to implement an efficient version of Conway's *Game of Life* [9] able to cope with large, potentially sparse simulations. Therefore, we ask them to pay attention to memory usage optimization, by using their own, low memory footprint data structures for computations. EASYPAP allows kernels to use arbitrary data structures for computations. Such kernels simply have to update the current image when a graphical refresh is needed.

In addition, students have to develop a lazy evaluation algorithm that avoids computing tiles whose neighbourhood was in a steady state at the previous iteration. Once they end up with an effective Pthreads or OpenMP lazy variant, students can look at the tiling window to make sure that areas where "nothing changes" are not computed.

Finally, students extend their implementation in order to cope with distributed architectures by using MPI. They learn how to exchange ghost-cells between MPI processes, including meta-informations regarding the state of tiles (steady or lively). The whole code is less than 150 lines. EASYPAP helps by integrating the `mpirun` process launcher and only displaying the main window of the master process by default.

For debugging purposes, EASYPAP can display all the windows for each process. The following command launches two MPI processes executing the `mpi_omp` variant in debugging mode.

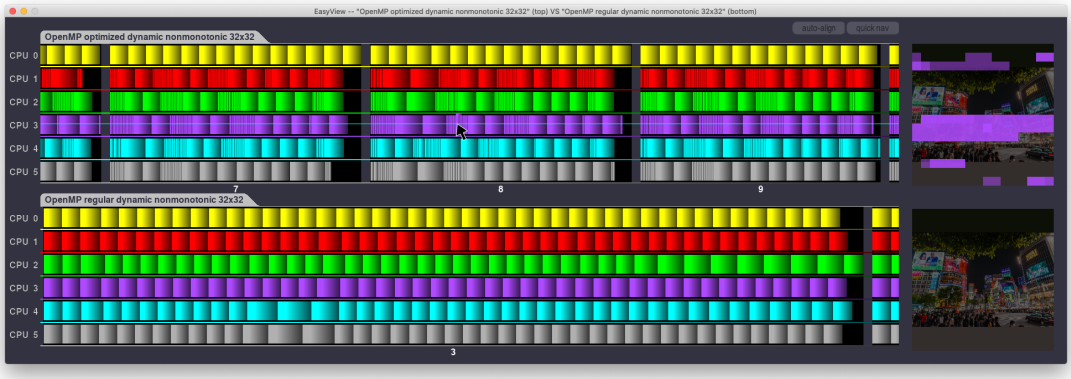



Figure 10. Comparison of two execution traces of the `blur` kernel using EASYVIEW. The bottom trace corresponds to the execution of a basic OpenMP implementation using uniform tiles. The top trace corresponds to an optimized OpenMP version where conditional code was removed from inner tiles. This later version is approximately 3 times faster in this setup (iteration 3 with the basic version is as long as iterations [7..9] with the optimized version).

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

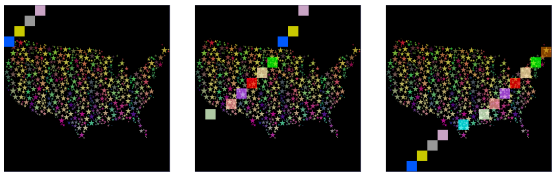


Figure 12. EASYVIEW allows to visualize the wave of tasks moving forward during the execution of code depicted in Fig. 11. These three screenshots were taken while moving the mouse from left to right over the Gantt window.

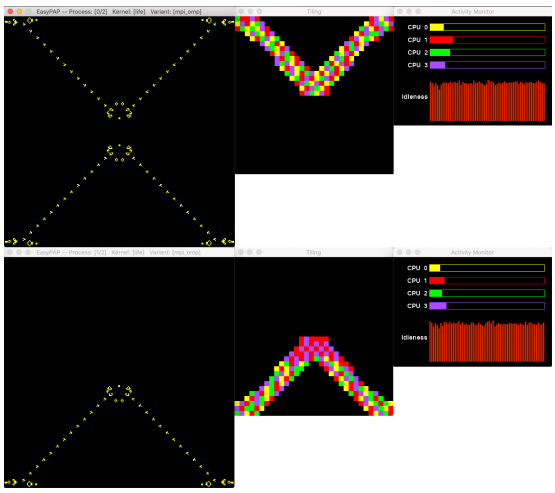


Figure 13. When launched in debugging mode, monitoring windows of every MPI process show up and help to visualize which area is processed by each of them.

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

The monitoring windows (Fig. 13) reveal that each process contains 4 threads and works on half of the image. Most importantly, since the sparse dataset consists in planers evolving along the diagonals of the image, we can check that only tiles located near diagonals are computed.

### E. Discussion

Since we introduced EASYPAP in our lab sessions, in 2018, it is clear from our side that students find parallel programming much more attractive and fun. Graphical tools make their debugging sessions less painful and more effective. The fact that it took postgraduate students less than a dozen hours to come up with an efficient MPI+OpenMP implementation of the *Game of Life* kernel using lazy evaluation (see Section III-D) impressed us. This highlights the importance of allowing students to

quickly prototype preliminary variants of the code and analyze their parallel behavior interactively.

On the downside, EASYPAP provides the students with an integrated environment where all the low-level details of configuration, compilation and initialization of various components are hidden. So it is necessary to conduct more conventional lab assignments as well, involving writing small applications from scratch to show student how a complete OpenCL program looks like for instance.

## IV. RELATED WORK

There have been many contributions to the field of developping programming environments for teaching parallel programming [4], [6].

Like the authors of [4], we are convinced by the pedagogical benefits of using *exemplars*. EASYPAP is also built around the notion of exemplars that students can parallelize using multiple paradigms.

Regarding visualization, we adhere to the same philosophy as the ParaVis [6] and TSGL [5] efforts, which provide easy-to-use C/C++ interfaces to visualize 2D animations produced by parallel computations. These libraries are versatile and can be interfaced with almost any existing 2D simulation. EASYPAP follows a different approach by providing an integrated educational framework with monitoring and trace exploration capabilities, experience automation and plot generation assistance.

Many outstanding tools have been developed to visualize and analyze execution traces, such as Aftermath [7], Grain Graphs [11], Intel Vtune Profiler [1], TAU [13], Vampir [10] or ViTE [3]. We think EASYPAP represents a smooth and attractive first contact with trace analysis tools, before beeing introduced to more complex ones. An original aspect of both EASYPAP and EASYVIEW is that they establish a graphical link between computations (i.e. tasks) and their associated data (i.e. image tile).

## V. CONCLUSION AND FUTURE WORK

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

Currently, EASYPAP only partially supports OpenCL: users can observe animated output of kernels, but monitoring and trace exploration are not yet implemented. These features will soon be developed by leveraging OpenCL profiling events. In a near future, we also intend to further extend the EASYVIEW trace explorer to integrate per-task cache usage information using the PAPI library [14].

## REFERENCES


[1] "Intel Vtune Profiler." [Online]. Available: https://software.intel.com/en-us/vtune

[2] "SDL: Simple directmedia layer." [Online]. Available: https://www.libsdl.org

[3] "ViTE: Visual trace explorer." [Online]. Available: http://vite.gforge.inria.fr

[4] 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.

[5] 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.

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

[7] 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.

[8] ETP4HPC, "Strategic research agenda," 2017. [Online]. Available: https://www.etp4hpc.eu/pujades/files/SRA%203.pdf

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

[10] 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.

[11] 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.

[12] R. Namyst and P.-A. Wacrenier, "The EASYPAP web site," 2018. [Online]. Available: https://gforgeron.gitlab.io/easypap/

[13] 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.

[14] 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.