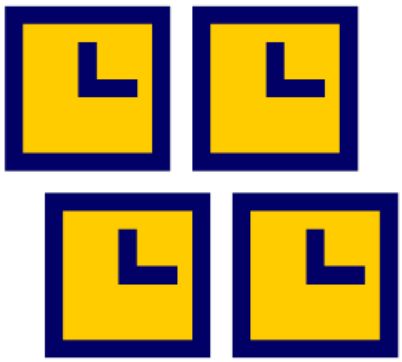


P-SOCRATES

Parallel Software Framework for Time-Critical many-core Systems

Deliverable type	Report/Prototype
Deliverable name	P-SOCRATES evaluation report
Deliverable number	D1.4
Work Package	1
Responsible partner	ATOS
Report status	Final
Dissemination level	Public
Version number	v1.2
Due date	31 December 2016
Actual delivery date	31 December 2016

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement nº 611016

P-SOCRATES Partners

Name	Short name	Country
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO	ISEP	Portugal
BARCELONA SUPERCOMPUTING CENTER – CENTRO NACIONAL DE SUPERCOMPUTACION	BSC	Spain
UNIVERSITA DEGLI STUDI DI MODENA E REGGIO EMILIA	UNIMORE	Italy
EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZURICH	ETH Zurich	Switzerland
EVIDENCE SRL	EVI	Italy
ACTIVE TECHNOLOGIES SRL	AT-ITALY	Italy
ATOS SPAIN SA	ATOS SPAIN SA	Spain

Project Coordinator
Dr. Luis Miguel Pinho
Email: lmp@isep.ipp.pt

Project Manager
Dra. Sandra Almeida
Email: srca@isep.ipp.pt

Contact information
CISTER Research Centre
Instituto Superior de Engenharia do Porto (ISEP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto, Portugal

Phone: +351 228340502
Fax: +351 228321159

Document History

Version	Date	Author	Description
0.1	18/11/2016	ATOS	ToC
0.2	25/11/2016	ATOS	First release (draft)
0.3	30/11/2016	UNIMORE	Initial contrib. to ES section
0.4	02/12/2016	BSC	Initial contrib. ESA section
0.5	02/12/2016	ATOS	Initial contrib. pCEP section
0.6	07/12/2016	ATOS	Updated sections 2, 3, 4
0.7	13/12/2016	BSC	Updated ESA section
0.8	15/12/2016	UNIMORE	Updated ES section
0.9	19/12/2016	ATOS	Release version
1.0	23/12/2016	ATOS	Final Release
1.1	27/12/2016	ISEP	Review
1.2	30/12/2016	ATOS	Final Release (Final)

Table of Contents

1 Summary

This deliverable compiles the final experimentation tests conducted over the three use case applications defined in P-SOCRATES. The P-SOCRATES Software Development Kit (SDK) has been used in order to enhance the parallelization strategies applied to the applications. A round of evaluations has been done using this SDK to compare the performance against other software setups, in particular when a non-sequential strategy is followed using the standard SDK provided by the manufacturer.

2 Purpose and scope

2.1 Purpose

This deliverable consists of a report summarizing the evaluation results on the three use cases considered in the project, analysing applicability and effectiveness of the proposed P-SOCRATES techniques. The report is accompanied by the final prototype of two of the use cases (the third use case is covered by a NDA). Within Task 1.4 the integration of WP2 to WP5 contributions into the applications has been coordinated to perform the final evaluation so that the performance opportunities of current many-core embedded processors can be fully exploited while providing trustworthy real-time guarantees. The final evaluation campaign was conducted over the use cases for their sequential and parallel versions (both using and not using P-SOCRATES SDK).

2.2 Scope

The scope of this document is *public*. However, some of the information covered by this deliverable may be limited exclusively to the partners of the P-SOCRATES project for a given period of time. The reason is that some of the contents of this deliverable may be susceptible to be published in conference proceedings or journals.

2.3 Abbreviations and Acronyms

The list of abbreviations and acronyms used in this document is shown below.

Abbreviation Acronym	Description
CEP	Complex Event Processing
COTS	Commercial-Off-the Shelf
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DoW	Annex I - "Description of Work"
ESA	European Space Agency

HP	High Performance
IG	Interference Generators
LP	Low Performance
MEET	Maximum Extrinsic Execution Time
MIET	Maximum Intrinsic Execution Time
MOET	Maximum Observed Execution Time
MPPA	Massively Parallel Processor Array
MS	Milestone
NDA	Non-Disclosure Agreement
SDK	Software Development Kit
SOLCEP	Smart Object Lab CEP
TDG	Task Dependency Graph
WCET	Worst-case Execution Time

2.4 Structure of this document

This document is structured in sections ordered to present the information gradually:

- Summary: provides a brief summary about the work conducted in Task 1.4.
- Purpose and Scope: explains the purpose, scope, and structure of this document.
- Section 3: Describes the HW and SW used during the evaluation campaign.
- Section 4: Presents the results and conclusions extracted out of the evaluation of the three applications.
- Section 5: Conclusions of the evaluation report.

3 Experimentation Environment

3.1 The HW setup

The hardware board used during the experiments is the Kalray Bostan MPPA[1] (Figure 1), which is the second generation of Kalray's manycore processor family. Bostan brings an ASIC-level of performance (high computing, low power consumption and real-time processing) with full programmability. Bostan runs 288 C/C++ programmable cores, optimized for networking and storage applications, and includes high speed interfaces like 80GE and PCIe x16 lane Gen3 directly connected to the large matrix of 288 cores and 128 crypto co-processors, whose core is a 64-bit Very Long Instruction Word (VLIW).

The processor is divided into clusters: 4 clusters which have more powerful cores, intended to perform the input-output with the outside (IO cores) and 16 computing clusters, each with 16 "working" cores. In the model considered in the project, applications start executing in the IO cores, and offload parallel computation into the computing clusters.

The hardware manufacturer provides its own development environment for this board, the Kalray AccessCore™ SDK, providing a standard C/C++/Fortran based environment including all the tools to quickly develop, debug, and optimize high-performance applications for the MPPA processor.

Figure 1: Kalray Bostan MPPA board.

The project started working with the first generation of Kalray manycore processor family, the Kalray Andey MPPA, composed of an array of 16 clusters and 4 I/O subsystems, themselves connected by two NoCs, whose core is a 32-bit VLIW. Since this version was discontinued by the manufacturer, the consortium had to move forward and adapt the current work to the next generation of the board, i.e., the Kalray Bostan MPPA.

[1] Kalray Bostan MPPA: http://www.kalrayinc.com/kalray/products/#processors

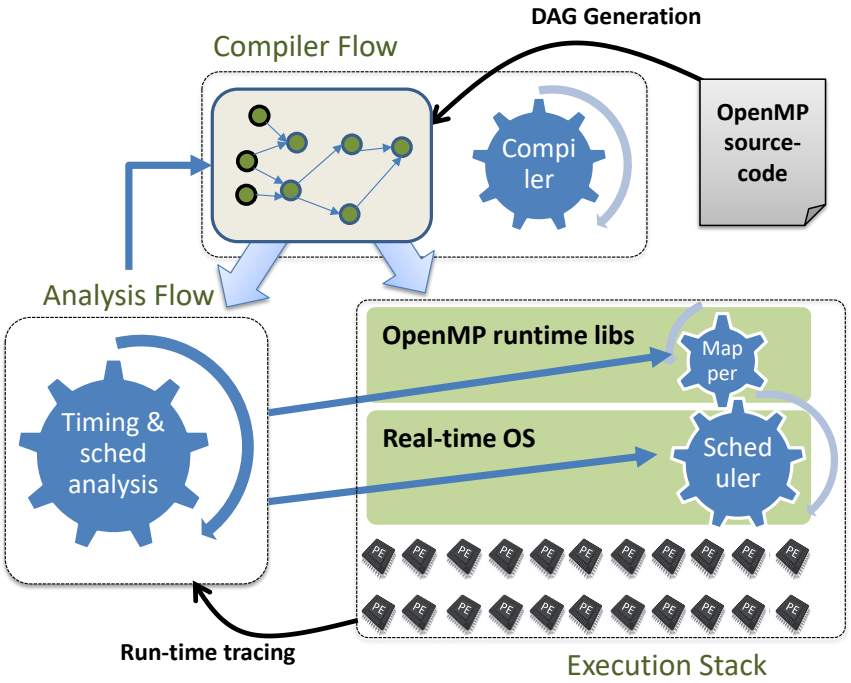


Figure 2: P-SOCRATES SDK.

3.2 The SW Setup

There are three different SW setups which will be used to evaluate the use case applications:

- **Sequential** – In sequential mode the application does not take advantage of the many-core architecture embedded in the Kalray board. On the contrary, the application does not spawn any cluster cores, leveraging the execution just to one core (IO core in the Kalray nomenclature).
- **Parallel Kalray baseline** – In baseline mode the application has been parallelized using the annotations and instructions provided by the HW manufacturer, using the Kalray AccessCore SDK 2.2.2. In one of the use cases it was also possible to compare with a low-level thread-based parallel baseline.
- **Parallel P-SOCRATES** – The consortium ported the P-SOCRATES SDK to the Kalray Bostan MPPA board (the SDK had been initially targeted to the previous version of the processor). Figure 2 (extracted from Deliverable D1.2.2 [1]) is included as a quick reference to the main components that comprise the P-SOCRATES SDK: the compiler, the timing & schedulability analysis tools, the OpenMP runtime libraries and the real-time OS. In this deliverable, the evaluation focus on the use of the P-SOCRATES SDK with the dynamic scheduling approach (see Deliverable D4.3.2 [2]), where better average performance is expected, due to the higher adaptability. The differences between the dynamic and static approaches of P-SOCRATES is explored in Deliverable D4.4 [3], since the use of the static approach (targeting higher predictability) requires performing per-task timing analysis.

3.3 Evaluation Scope

Following the evaluation procedure presented in Deliverable D1.2.2 [1], in this final evaluation campaign a complete assessment is enforced for each of the use case applications, on the three different implementation: sequential, parallel baseline and P-SOCRATES.

P-SOCRATES provides three use case applications: an online semantic intelligence tool, a pre-processing sampling application for infra-red detectors, and a parallel Complex Event Processing Engine. From the original implementation of these applications, which were developed in a sequential manner, the project consortium created two new parallel versions: one using the baseline approach, and one using the P-SOCRATES SDK.

Table 1 is used as a reference to perform the experiments on each of the use cases and extract relevant conclusions. Note that the efficiency of the analysis is dealt with in Deliverable D4.4 [3].

Table 1: Evaluation Metrics.

Appraisal criteria	Metrics
Improvement of performance	Reduction/Increase of average response time
	Reduction/Increase of worst case response time
	Variance of response time distribution
	Performance per watt (to normalize) between platforms
Efficiency of the analysis	Ratio of worst case response time
Efficiency of the run-time	Reduction/Increase of the memory footprint
	Reduction/Increase speed-up
	Ratio of parallelism
	Reduction/Increase of task granularity
Complexity of parallel code	Total number of lines of code against sequential
	Total number of used functions against sequential
	Extra functionality against relevant computing platforms

4 Use Cases Evaluation

4.1 Online semantic intelligence tool

ADmantX™ is a software service for publishing smart advertisements on web pages. It was developed by Expert Systems, located in Modena, Italy. The main task performed by ADmantX™ follows (Figure 3): it receives the URL of a webpage currently navigated by a user, whose profile and preferences are known in advance because he "logged in" onto the website. Based on the text contained in the webpage, it understands which “categories” the text matches, and communicates it back to the calling webpage, which will use them to choose the most suitable advertisement to show according to the user's profile. What is important here is that ADmantX™ is part of an "advertisement" auction against other tools providing the same service. ADmantX™ must perform this analysis within 10 milliseconds: if it "loses the auction", the company does not get an income for that service invocation.

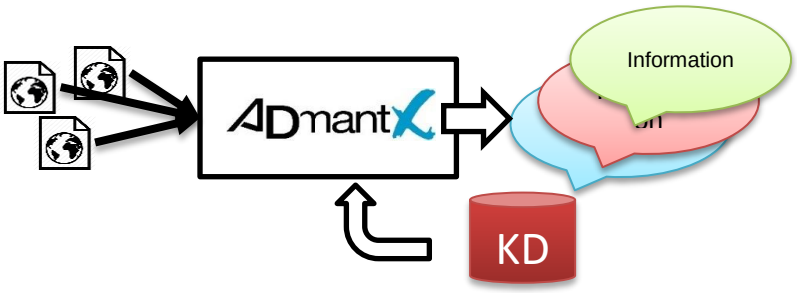


Figure 3: Software architecture of the online semantic tool.

The Information extracted is in the form like *"this webpage text matches 80% with the concept of 'Barcelona – the city'"*, or *"This texts expresses happiness (60%)"*.

To perform this categorization, ADmantX™ performs the page-to-category (or user-to-profile, respectively) match by using its internal semantic engine, which in turn relies on a Knowledge Database (KD) where information extracted from the web page is searched and categorized. These *lemmas* are strings made of multiple words, such as "Alan Turing", or "Bank", or "The capital of Italy", but of course "Alan", "Turing", "Alan Turing was a mathematician" or "Capital" can also be a lemma.

Every time a webpage is parsed by the system, potentially hundreds-to-thousands of searches are performed. After a first analysis, we found that the most time-consuming portions of code (i.e., functions, or portion of functions), and the most amenable to parallelization is the so-called *Disambiguator*, which analyses the text according to a given language, and builds a network of concepts for fast information retrieval.

4.1.1 Memory issues, and batched parallelization strategy

The main issue that we faced is that the Knowledge Database (KD) has a high memory footprint of approximately 2.5 MB (even after being re-designed from scratch for low occupancy), while on each cluster of the Kalray MPPA accelerator we have less than 2MB of shared scratchpad memory (which needs also to consider the footprint of the Erika Enterprise operating system and OpenMP runtime). After a first analysis, we found the following distribution of the "initial letter" of each lemma in the KD [2] (Figure 4). Since they are irregularly distributed, we decided to assign each letter to a cluster.

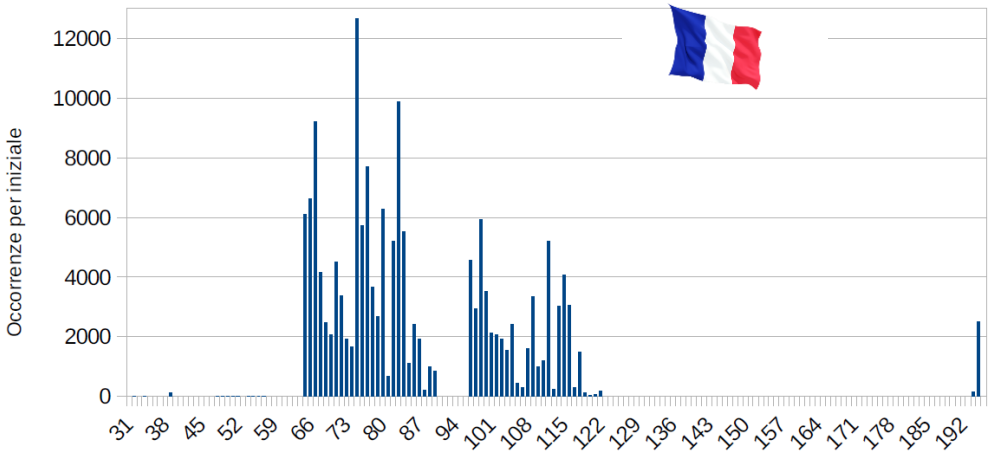


Figure 4: Distribution of the first letters (ASCII code) of lemmas in KD.

This memory constraint heavily affects the parallelization strategy. Since KD is alphabetically partitioned onto accelerator's clusters, and we treat each cluster as a "computations island" (so that it is not required to perform inter-cluster memory accesses), the parallelization scheme needs to be as described next (and represented in Figure 5).

IO cores host the web service that receives the pages to analyse, and split the page onto batches of searches according to Expert System proprietary semantic algorithm. Searches are then prioritized, and batched alphabetically. Then, batches of searches are offloaded using the OpenMP `target` directive to the cluster that hosts the subset of the KD matching the first letter. Searches are case-sensitive.

[2] The KD used for the evaluation is in the French Language.

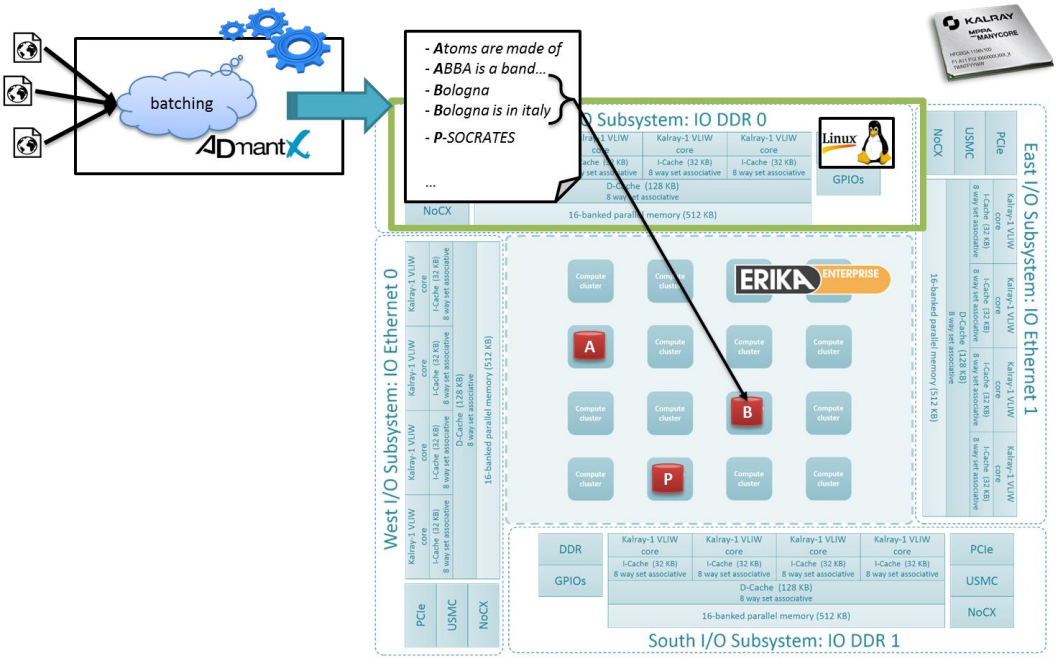


Figure 5: Application parallel partitioning strategy into batches.

4.1.2 Intra-cluster fine-grained parallelization with OpenMP tasks

Inside a single cluster, a batch of searches is performed, as shown in Figure 6.

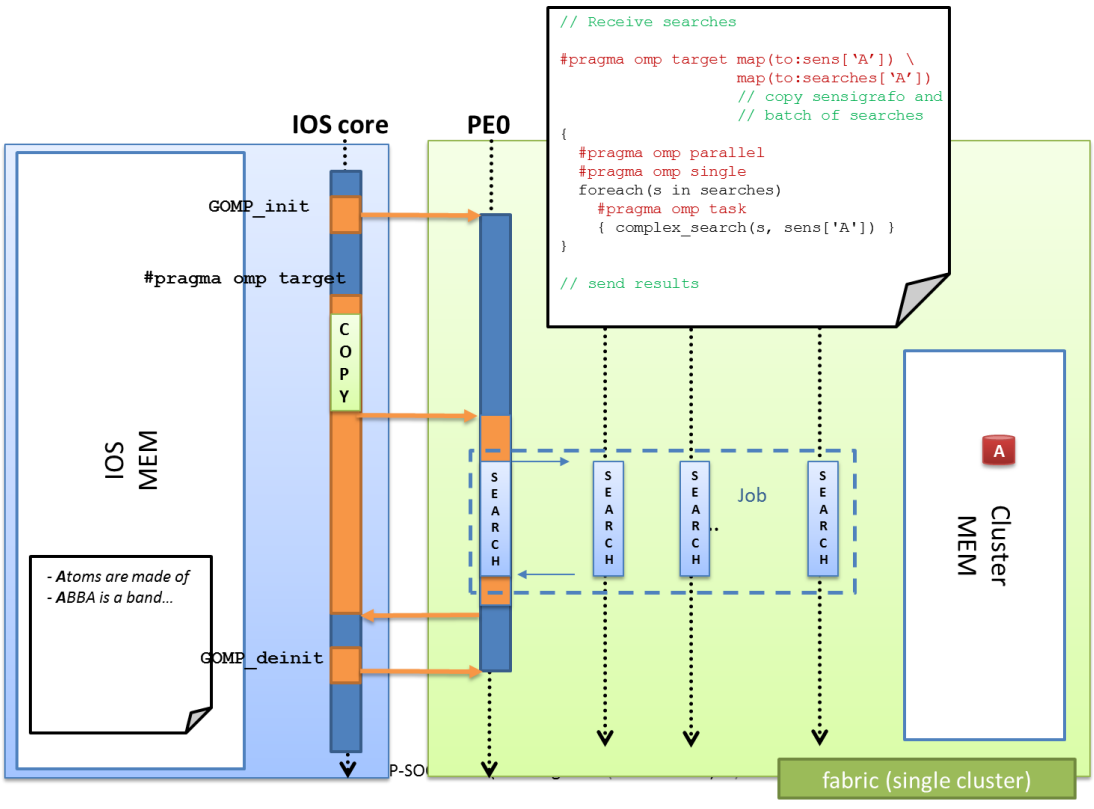
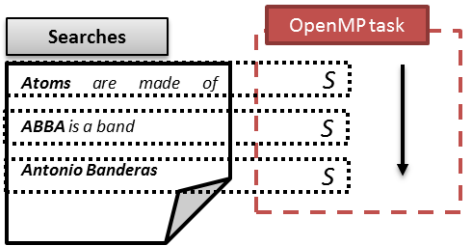


Figure 6: Timeline of application offloading onto the many-core fabric.

In this implementation, we have one thread producing the work (as per the usage of the OpenMP single construct), that is, the multiple OpenMP tasks, and all the threads of the parallel region consuming it. This is a typical, flexible yet lightweight, parallelization strategy employed for parallel accelerators, and that is used in the model of P-SOCRATES.

It is important to note that searches are very fine grained (a search takes in the order of 1 microsecond), hence each one alone is not worth the cost of the parallelism overhead introduced by the runtime stack (even the optimised lightweight tasking library of P-SOCRATES [4]). Indeed, when we first run this scheme, we experienced a more than 2 times slowdown against the sequential version, for each OpenMP task/search[3]. As this was clearly not acceptable, we then employed a chunking strategy, where more than one search is used within each OpenMP task, to increase its size, as shown in Figure 7.




```
#pragma omp target map(to:sens['A']) \
                   map(to:searches['A'])
                   // copy sensigrafo and
                   // batch of searches
{
  #pragma omp parallel
  #pragma omp single
  foreach(s in searches / CHUNK)
    #pragma omp task
    {
      for(i<CHUNK)
        complex_search(s[i], sens['A'])
    }
}
```

Figure 7: Chunking of searches onto OpenMP Tasks.

We can tune this task granularity using the `CHUNK` parameter, e.g., in this example `CHUNK` = 3. Intuitively, increasing the task granularity has the effect of reducing the number of tasks. Since we have only 16 cores/threads per-cluster, this might lead to corner cases where there is not enough work to spawn, and underutilize clusters' computing power.

The main effect of such a parallelization strategy, is that we spawn independent searches in parallel. In this use-case and under this parallelization strategy, this means that there are no explicit dependencies among parallel tasks that can be captured by using the OpenMP depend clause. The semantic analysis application represents a specific case of an application, where dependencies among tasks are only implicit: the limit on the number of cores in the system (e.g., 16 per-cluster) causes serialization effects among tasks (as exemplified in Figure 8 for 4 tasks and 2 cores.

[3] We measured the time taken for one in-cluster parallel region, subtracted the overhead of the parallel construct (that is known because the OpenMP runtime was profiled – see Deliverable 6.4 [2]) and divided for the number of searches.

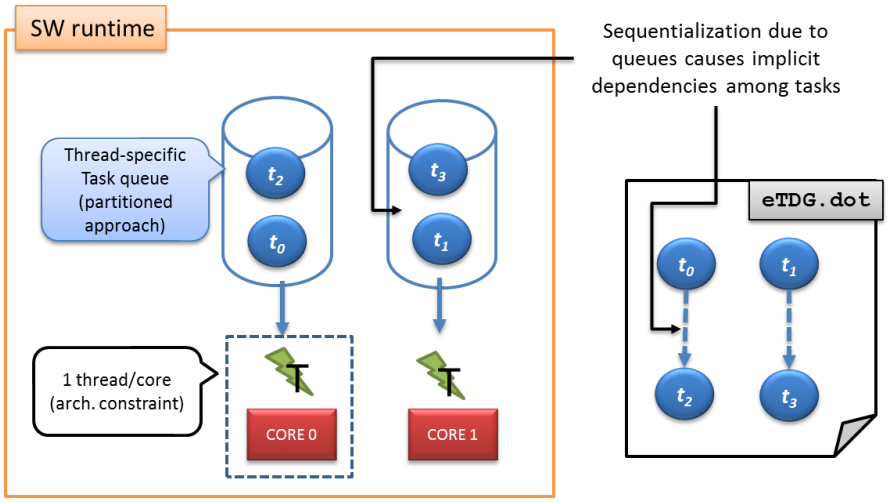


Figure 8: Serialization of four TDG nodes on a two-queues system.

4.1.3 Performance improvement and efficiency of the runtime

We address performance using first the **average** as a metric. First, this application is not hard real-time, and there is an acceptable trade-off that allows some searches not to meet the 10 milliseconds if on average there is an increase of successful searches. Second, this specific application is heavily data-dependant. The longest response time happens when a search actually **fails**, that is, the lemma is not present in the KD. Nevertheless, we also address maximum observed and the deviation [4] of the searches' times.

In Figure 9, we show how parallel performance (speedup against sequential implementation) varies with the number of searches-per-task (CHUNK) and with the number of threads, assuming that in total 256 searches need to be performed in parallel.

On the left side of the chart, we have the fine-grained tasks that lead to the *naive* case from which we started: 1 search-per-task (256 tasks) leads to 0.4X speedup (2.5 slowdown). On the rightmost side, we see the opposite situation, where all the 256 searches are chunked in a single task. This latter is actually a sequential execution, with a small slowdown against the sequential implementation because of the tasking runtime overhead.

[4] The *standard deviation* quantifies the amount of variation of a set of data values. A low value indicates that elements of a data set tend to be close to the average the set, while a high value indicates that data elements are spread out over a wider range of values.

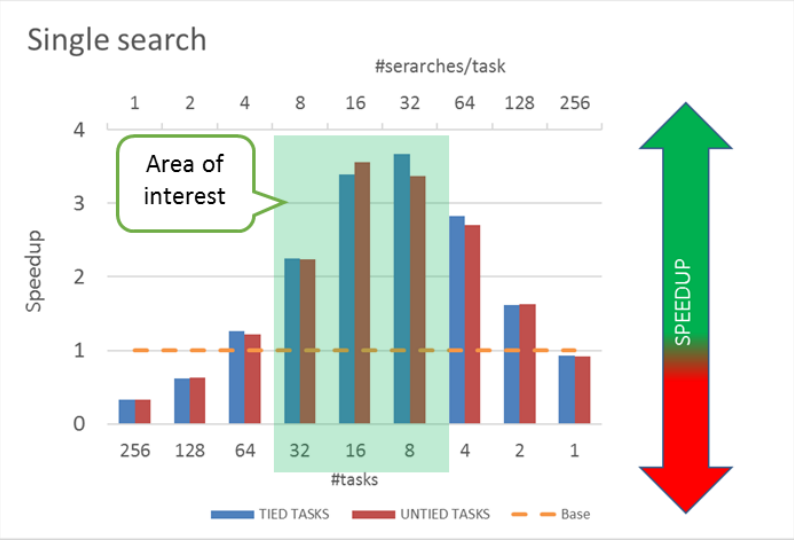


Figure 9: Parallel performance scaling with CHUNK parameter (256 searches).

In the middle of the chart, we notice a behaviour where (from left to right) we first have a performance improvement due to the increasing coarseness of tasks, and then a performance deterioration because we cannot exploit all the cores available in the cluster (note the lowermost X axes). We highlighted an "area of interest" where we are "wisely" using the multi-core cluster, that is, we can exploit the computational power of 16 cores with negligible runtime overhead (tasks are coarse-grained enough).

Since the P-SOCRATES OpenMP runtime also supports *untied* OpenMP tasks, other than *tied* tasks, we also show the performance for that specific parallelization scheme. Note that a tied task can only be executed by the thread that created it (thus cannot migrate to a different thread, when queued, even if the other thread is idle) while untied tasks can be migrated, so more flexible. In this case, there is no noticeable performance variation, because the difference shows more in nested task parallelism which does not occur in this use case. For a more detailed description of tied and untied tasking model, and the differences existing among them, we refer the reader to Deliverable D4.3.1 [5].

We also wanted to measure the parallel performance against other parallelization strategies. To this end, we implemented the algorithm using "PThreads-style" loop-based parallelization with a static partitioning of iteration among threads. This strategy is known for exhibiting the minimal overhead possible, at the price of lower programmability. For this reason, this strategy represents the parallel-best. It is depicted in Figure 10, where we also show the performance for OpenMP tied tasks strategy by using the native Kalray SDK and runtime libraries [5].

[5] Kalray runtime environment does not yet support untied OpenMP tasks.

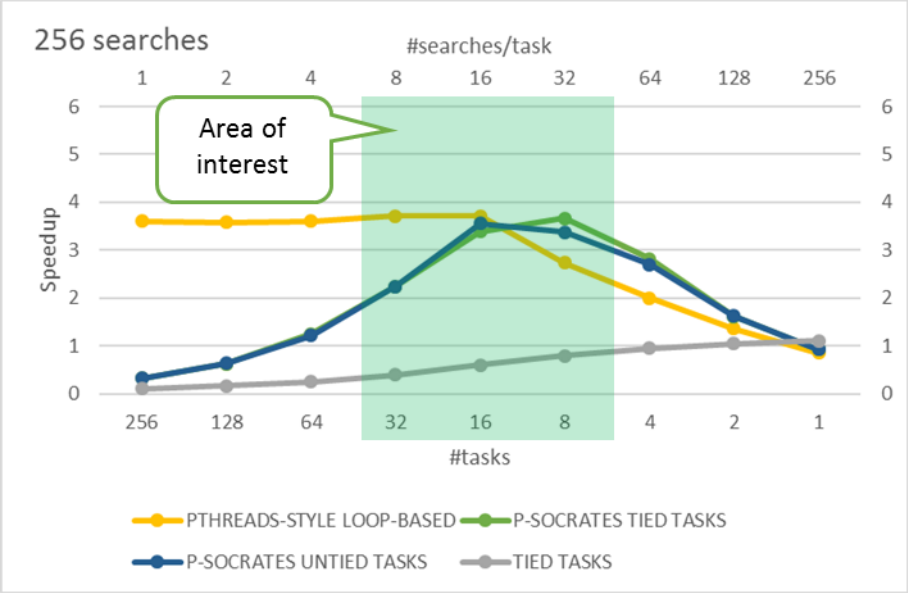


Figure 10: Task performance comparison against parallel loops and Kalray native tasking (256 searches).

While our support outperforms Kalray native SDK by 3-4 times, we are still under the reference best-case loop-based strategy, in the leftmost part of the chart/design area, with low-granularity tasks. Nevertheless, in the mid-range, we achieve similar speedup as this parallel-best baseline. As mentioned previously, this use-case does not benefit from the task migration, hence there is no difference between untied and tied tasks.

Figure 11 shows how increasing the number of searches (hence the number of threads for small CHUNKS - left, and the task size for low number of threads - right) enables us outperforming baseline loop-based strategy for 32-64 tasks.

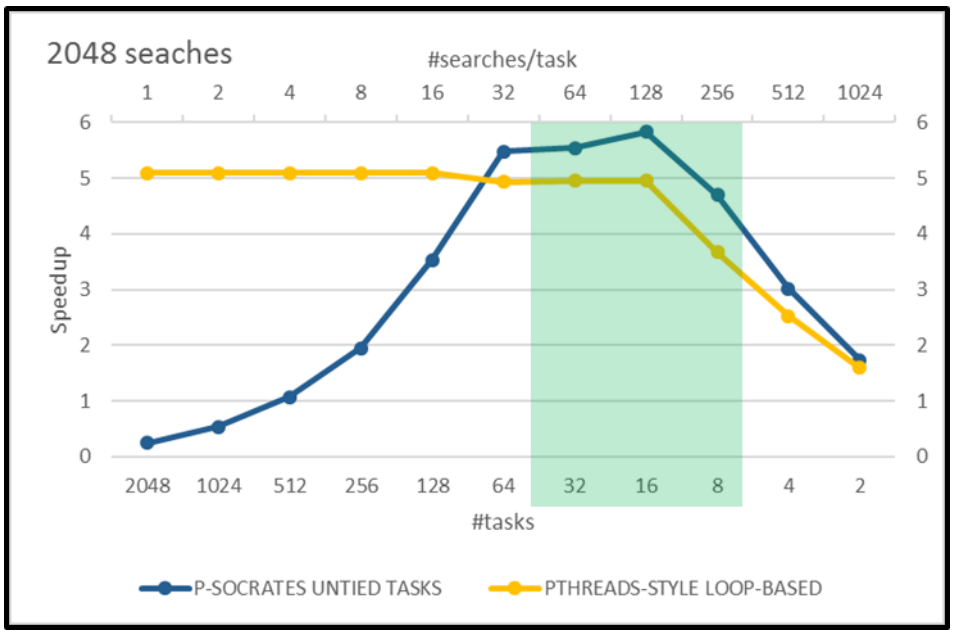


Figure 11: Task performance comparison against parallel loops.

4.1.4 Worst-case performance and standard deviation

In a dedicated set of experiments, we have extensively measured worst-case observed performance, and its variance against the average-case performance.

We executed the application a hundred times, collecting the time it took for executing both the full parallel region, and a single task. Table 2 shows the average, maximum (worst-case), and standard deviation, against the full set of experiments, of the time it takes to execute tasks made of 32 iterations (CHUNK = 32). Performance numbers for the "full" parallel region are substantially similar, and we do not show them. In this experiment, we again compare the P-SOCRATES SDK and the native Kalray SDK+runtime. Numbers are in nanoseconds.

Table 2: Average, maximum and standard deviation comparison (CHUNK=32).

	P-SOCRATES	Kalray
Average	11659	38963
Max	11687	39000
Standard deviation	9.75	19.12

As seen, there is not much difference between the average and the worst-case, and the standard deviation is almost two times smaller by using P-SOCRATES. These numbers are also in line with what we present in Figure 10 for average-performance.

We also performed the very same set of experiments with CHUNK = 64 (that together with CHUNK = 32 represents the "area of interest" of our application partitioning space). Results (Table 3) are similar, despite they are scaled up of a factor of ~2.5, reflecting the different coarseness of tasks.

Table 3: Average, maximum and standard deviation comparison (CHUNK=64).

	P-SOCRATES	Kalray
Average	27121	69032
Max	27187	69375
Std dev.	17.46	30.57

With respect to the standard deviation, the P-SOCRATES stack outperforms the native SDK, here by a factor of ~1.5 times. Nevertheless both values are very small.

4.1.5 Complexity of parallel code

In this section, we compare the complexity of the code of the application parallelized with our strategy against the version running on top of the Kalray MPPA SDK. We compare the version of the code parallelized with our SDK against both the sequential version and the parallel version of the code on top of Kalray native OpenMP frontend.

High-level programming languages such as OpenMP provide the right expressiveness and abstraction to hide the complexity of the underlying platform. Especially, the adoption of OpenMP `parallel` and `task` constructs within computing clusters significantly reduces not only the

number of code lines, but also, e.g., the number of **code functions** necessary to express parallelism with most of the "traditional" parallel frameworks. As an example, the following snippets of code shows the very same portion of code parallelized with OpenMP tasks, and POSIX Threads (PThreads).

```c
#include <pthread.h>

void *thrd_body(void *arg)
{
  // Application code. Return some value
  return (void *) 0x0;
}

int main()
{
  pthread_attr_t myattr;
  pthread_t thethreads[16];
  int returnvalue, parameter = 1234;

  /* initializes the thread attribute */
  pthread_attr_init(&myattr);

  /* creation and activation of the new thread */
  for(int i=0; i<16; i++)
    pthread_create(&thethreads[i],
      &myattr, thrd_body,
      (void *)&parameter);

  /* the thread attribute is no more needed */
  pthread_attr_destroy(&myattr);

  /* wait the end of the threads we just created */
  for(int i=0; i<16; i++)
    pthread_join(thethreads[i], &returnvalue);

  printf("main: returnvalue is %d\n",
      (int) returnvalue);

  return 0;
}
```

Code parallelized with PThreads

```c
int main()
{
  int parameter = 1234, returnvalue;

  #pragma omp parallel num_threads(16)
  #pragma omp single
  for(int i=0; i<16; i++)
  {
    #pragma omp task firstprivate(parameter) \
                     shared(returnvalue)
    returnvalue = 0x0;
  } // End of parreg

  printf("main: returnvalue is %d\n",
      (int) returnvalue);

  return 0;
}
```

Code parallelized with OpenMP

Figure 12: Different parallelization strategies.

As shown in Figure 12 (left), in a lower-level mechanism such as PThreads, programmers must **manually extract** the parallel portions of code, and put them in a function whose signature must be

```
void *thrd_body(void *arg)
```

Additional code is then required to adequately instruct the runtime on how to treat the new thread (e.g., with the `pthread_attr_t`). It is clear how parallelizing an application with OpenMP is extremely simpler thanks to the adoption of code annotations, and it's also simpler to manage and debug. Note that one important achievement of the project is that using the higher-level of abstraction of OpenMP we achieve similar performance as the optimised low-level code.

We also found that one of the main limitations of the Kalray MPPA SDK is the lack of support for OpenMP 4.x `target` extension. By providing OpenMP 4.5 support, our SDK greatly simplifies

programmers' life, increases their productivity, and reduces the number of bugs in their code. The following pseudo-code snippet shows how we parallelized the semantic analysis software.

```
#pragma omp target map(to:sens['A']) \
                   map(to:searches['A'])
                   // copy sensigrafo and
                   // batch of searches
{
  #pragma omp parallel
  #pragma omp single
  foreach(s in searches / CHUNK)
    #pragma omp task
    {
      for(i<CHUNK)
        complex_search(s[i], sens['A'])
```

In order to assess the reduction of code complexity in our application against native support, we adopt the three metrics depicted in Table 4.

Table 4: Code complexity comparison of the three versions of online semantic application.

		Baseline Kalray (Sequential)	Baseline Kalray (Parallel)	P-SOCRATES (Tasks)
Complexity of parallel code	Total number of lines of code [6]	924	928	430
	Total number of used (C) functions	11	11	10
	Extra functionality against relevant computing platforms	Ref.	Ref.	- OpenMP target clause - OpenMP untied tasks

As shown in this table, the difference between the P-SOCRATES approach and the baseline Kalray SDK version is clear. The version parallelized by using the P-SOCRATES SDK has half as many lines of code as both the sequential and parallel code running by using the Kalray SDK. This improvement has been possible thanks to the adoption of the target pragma. The two versions that use the native SDK (sequential and parallel) differ by only the four lines of code (924 vs. 928) necessary to insert OpenMP pragmas for tasking, which is supported also by Kalray.

4.1.6 Evaluation of power consumption

We performed an extensive set of experiments to assess the power and energy consumed by the Kalray MPPA accelerator running the test-case. The Kalray MPPA System-on-Chip has a power

[6] Not including data from KD, that is hardcoded

envelope of approximately 8Watts that can go up to 15W under heavy workload conditions. It has four main power domains, as shown in Figure 13.

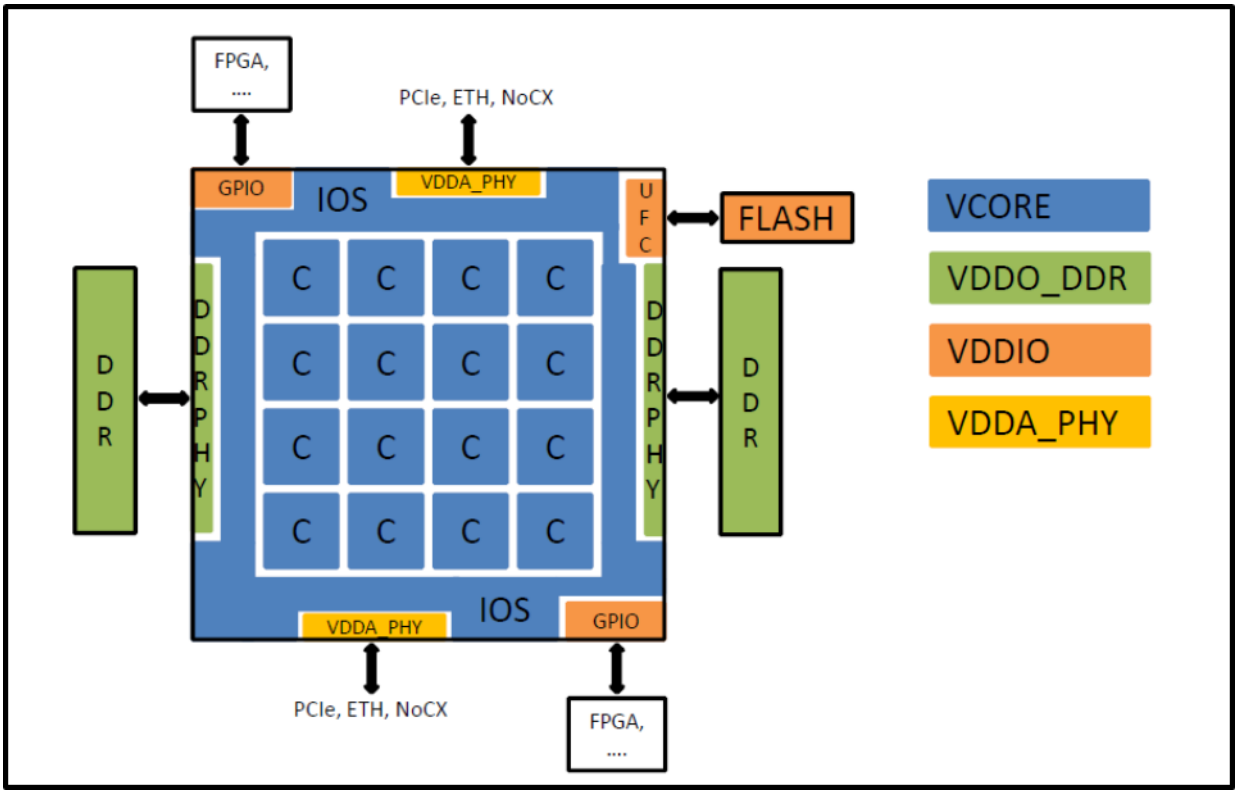


Figure 13: The four power domains of the Kalray MPPA fabric.

As illustrated in this figure, the main power domain, called VCORE, feeds both the IO cores and the accelerator fabric. There are three other power domains, which are connected to the GP-I/O peripherals, off-chip DDR and flash memory banks, respectively. In this use-case, we do not employ neither a flash memory, nor a GPIO, and the usage of the DDR memory is limited to the tiny (sequential) part of computation that is run on the host. For this reason, we focus only on the VCORE domain, which consumes on average 5 W out of the 8 of the whole chip. Table 5, taken from MPPA white sheets, provides more details on the four power domains.

Table 5: Kalray MPPA power domains and envelope.

Power Name		Operating voltage Range (mV)	Min (V)	Typ (V)	Max (V)	Typ Current (A)	Typ Power (W)
VDD_CORE		50	0.80	0.85	0.90	6	5
VDDIO		200	1.60	1.80	2.0	0.30	0.50
VDDO_DDR0	DDR3	150	1.35	1.50	1.65	0.35	0.50
	DDR3L	150	1.20	1.35	1.50	0.25	0.35
VDDO_DDR1	DDR3	150	1.35	1.50	1.65	0.35	0.50
	DDR3L	150	1.20	1.35	1.50	0.25	0.35
VDDA_PHY0		50	0.80	0.85	0.90	0.50	0.40
VDDA_PHY1		50	0.80	0.85	0.90	0.50	0.40

To collect power and energy measurements, we used a command line tool provided by Kalray called k1-power, which enables us to seamlessly run the use-cases the very same way as without measuring the power consumption (i.e., same application flags), and to store relevant information in text files inside a given folder. An example on how to use it follows.

```
$ k1-power --output=./<FOLDER>/ --gnuplot=pdf --traces_keep --
gnuplot_keep -- k1-jtag-runner --multibinary <APP_MULTIBIN_NAME> -
-exec-multibin=IODDR0:<IOS_EXECUTABLE>
```

We performed a set of experiments to assess the power consumed by the accelerator that is running the semantic use-case, in three configurations.

1. Our baseline is the sequential version running on the native Kalray Operating System (the topmost.
2. A version parallelized by using OpenMP tasks, running on the native Kalray OS + OpenMP runtime.
3. A version parallelized with OpenMP tasks, running on top of the P-SOCRATES stack, namely Erika Enterprise + our OpenMP runtime. In this case, we employ both tied and untied tasks

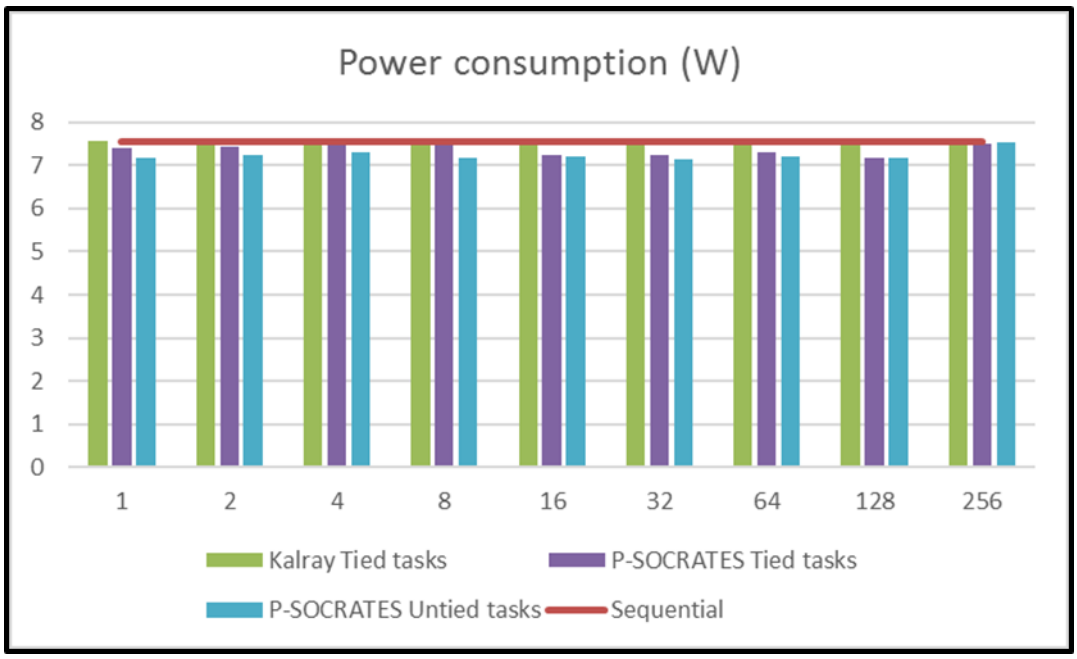


Figure 14: Power consumption of the online semantic application.

Figure 14 shows that in all of the configurations, we consume approximately 7.5 Watts, and there is a slight yet negligible power reduction by using our toolchain. Figure 15 shows how our SDK enables a reduction of the energy consumption by 20-25% in comparison to the sequential baseline, while Kalray native SDK consumes 10-12% more energy.

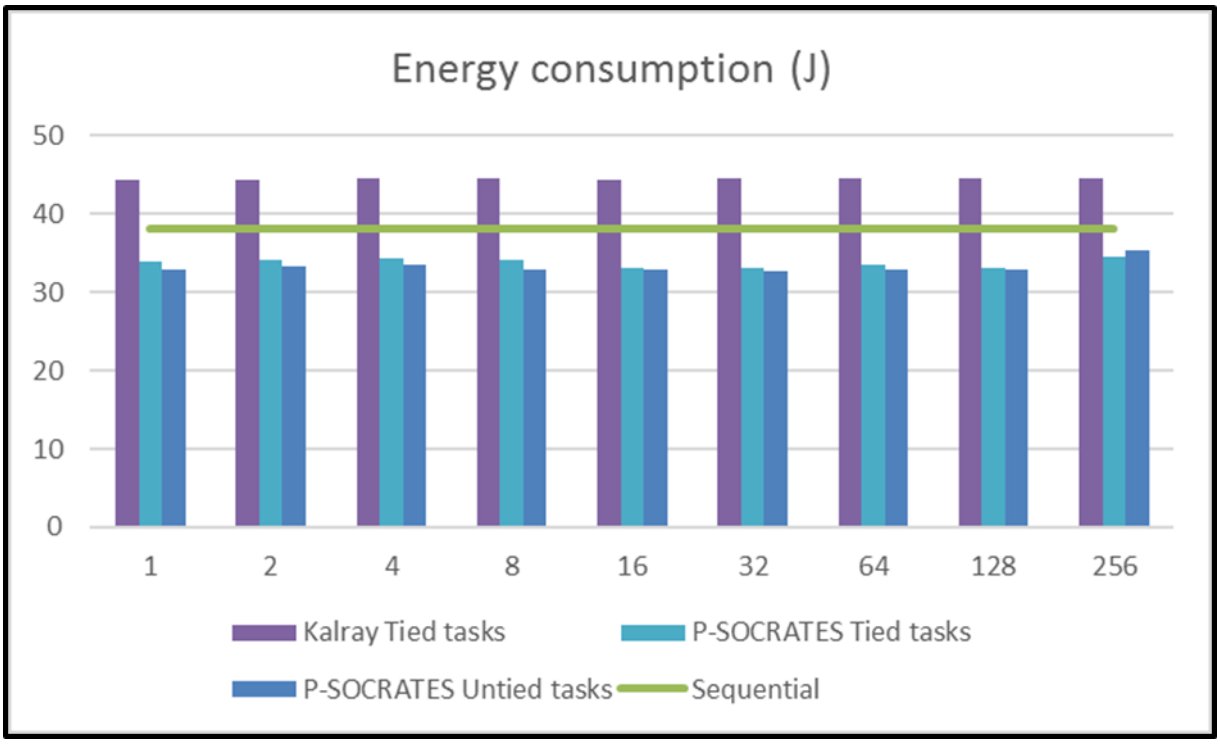


Figure 15: Energy consumption of the online semantic application.

4.2 Pre-processing sampling application for infra-red detectors

4.2.1 Short Description

The computational intensive part of the pre-processing sampling application for infrared H2RG detectors [7] is composed of seven stages executed sequentially, i.e. one after the other, within a loop iterating given number of times (in parenthesis, we specify the name of the C function implementing the corresponding stage):

1. The *saturation detection* stage (*detectSaturation*) detects when pixels go into saturation in order to reduce the readout noise.
2. The *super-bias subtraction* stage (*subtractSuperBias*) removes pixel-to-pixel variation by subtracting a bias frame from the detector's frame.
3. The *non-linearity correction* stage (*nonLinearityCorrection*) corrects the frame using a 4th order polynomial.
4. The first phase of the *reference pixel subtraction* stage (*subtractPixelTopBottom*) removes common noise by calculating the mean of odd and even pixels of the first and last 4 rows and subtract it from the odd and even pixel of the frame.
5. The second phase of the reference *pixel subtraction* stage (*subtractPixelSides*) removes common noise by computing the average of lateral pixels.
6. The c*osmic ray detection* stage (*detectCosmicRay*) estimates the disturbances produced by cosmic ray.
7. The *linear least square fit* stage (*linearLeastSquaresFit*) detects these disturbances.

There is a last stage outside the loop, *final signal frame* (*calculateFinalSignalFrame*), which updates the frame in the required output format. Figure 16 shows the sequential execution of the seven stages defined above within a loop bounded by the *#Groups* parameter.

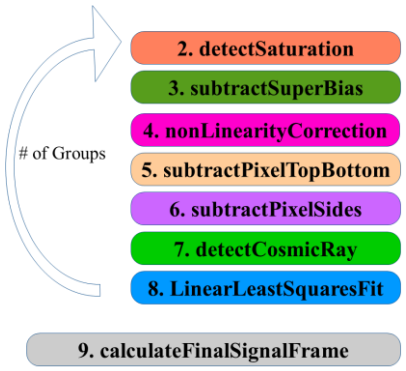


Figure 16: Pre-processing sampling application stages.

The application also includes extra stages executed at the beginning of the application, i.e. the *get frame* stage (*getCoAddedFrame*). This stage simulates the acquisition of a given number of readouts from the H2RG sensor frame and copies it into a 2048x2048 array structure. This function incorporates a *random* system call to simulate space radiation during the sensor data acquisition, which impacts negatively on the overall performance of the application. Clearly, this

[7] See deliverable D1.2.2 [1] for a complete description

impact is not representative of the physical data acquisition occurring into a real system [8]. To that end, the results presented in this project do not consider this first stage, and assume the usual process of sensor data acquisition overlapped with the computation of the previous acquired frame as shown in Figure 17.

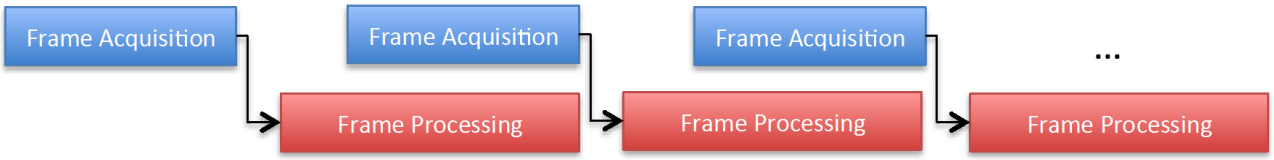


Figure 17: Pipeline parallelization strategy between the frame acquisition and frame processing.

4.2.2 Parallelisation Strategy on the MPPA

Deliverable D1.2.1 [6] presented the parallelisation approach, based on a *wave-front strategy* in which the frame is divided into *NxN* blocks, each being potentially executed in parallel by assigning each application stage to an OpenMP task. Moreover, in order to capture the data dependencies existing among the different stages, the tasks use the `depend` clause.

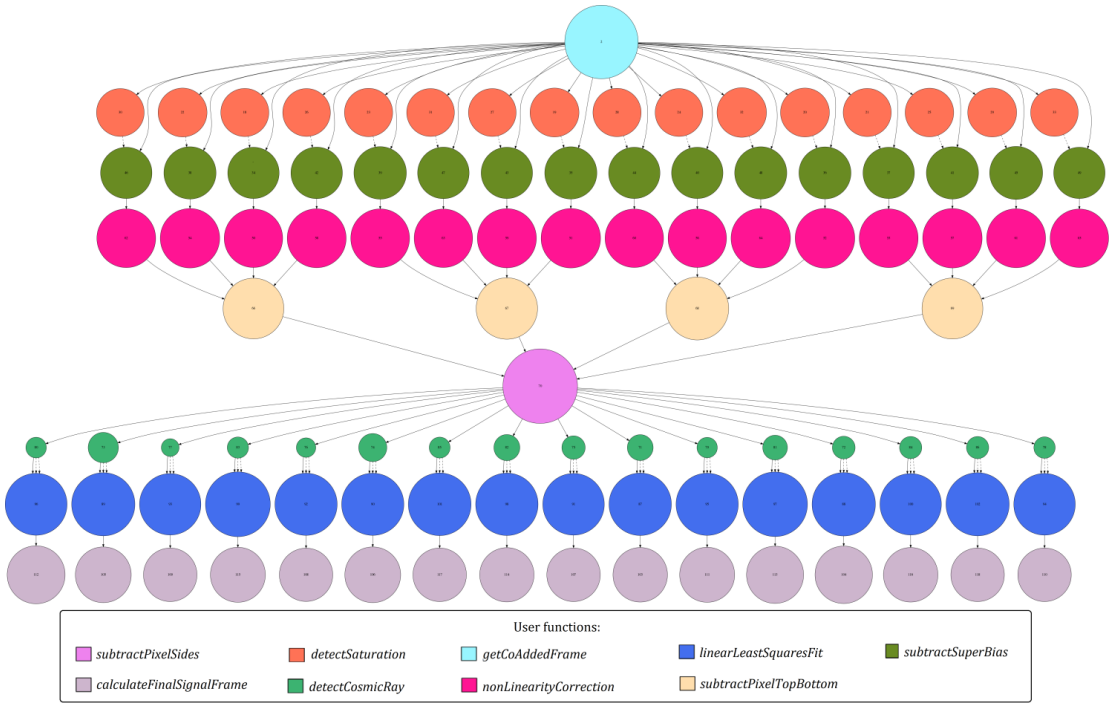


Figure 18: Data dependencies (in the form of TDG) among stages when dividing the frame in 16 blocks.

Figure 18 shows the data dependencies, in the form of a *task dependency graph* (TDG) extracted with the P-SOCRATES compiler, when dividing the frame in 16 blocks (N = 4) and assuming

[8] The overall performance of the application is limited by the full-frame readout of the H2RG sensor which rates from less than 0.1 Hz to 76 Hz according to http://panic.iaa.es/sites/default/files/H2RG_Brochure_rev6_v2_2_OSR.pdf

#Groups = 1. The figure also includes the *getCoAddedFrame* and *calculateFinalSignalFrame* functions.

In order to explore the performance opportunities on the Kalray MPPA processor architecture, we have developed a new parallel version of the application incorporating the `target` directive. The strategy adopted is to transfer to the accelerator devices of the selected architecture, a set block of frames to be processed in parallel.

Table 6 shows the memory requirements of input (*IN*), and input/output (*INOUT*) data dependencies required to compute each application stage as shown in Figure 18, under the following execution conditions:

- The column labelled as *Full Frame* shows the overall memory needed to store each data structure when the full frame is considered at each stage (sequential version).
- The column labelled as *64-block Frame* shows the memory requirements of each stage when processing in parallel 1, 2 and 4 blocks, assuming that the frame is divided in 64 blocks (parallel version). Notice that the size of some of the data structures remains the same when considering different number of blocks.
- The column labelled as *Blocks per stage* shows the overall memory requirements to execute a subset of stages within a cluster, assuming that 1, 2 and 4 blocks are simultaneously available in memory. This column uses the green and red colours to indicate when memory data structures respectively fit and do not fit within the cluster local memory.

The memory requirements to process in parallel functions *detect-Saturation*, *subtractSuperBias*, *nonLinearityCorrection* among 4 blocks is around 1 MB, thus fitting within a cluster memory (2 MB of capacity). For the *detectCosmicRay*, *linearLeastSquares-Fit* and *calculateFinalSignal-Frame* functions, 4 blocks cannot be processed in parallel as this operation requires 3 MB. In the case of 2 blocks, despite the memory requirement is less than 2 MB, the execution fails as an extra memory is needed for the operating system and run-time library. Therefore, only 1 block can be offloaded at the time.

The *subtractPixelTopBotton* function fits within the intra-cluster memory, as it requires a complete column of the frame of 1 MB to execute. It is important to remark that 4 extra blocks are required to be transferred to the cluster in which this function executes, i.e. 8 blocks correspond to a set of complete columns.

In the case of the *subtractReferencePixelSides* function, it cannot be executed within a cluster as it requires the full frame, and so it executes in the IO.

Table 6: Overall memory requirements of each application stage on the full frame and on each block, assuming that the frame is divided in 64 blocks.

In/Out data per Stage	Memory requirements						
		= 256 (64 blocks)			Accumulated blocks over stages		
	Full frame	1 block	2 blocks	4 blocks	1 block	2 blocks	4 blocks
2. detectSaturation		136 KB	272 KB	544 KB	264 KB + 320 bytes	528 KB + 320 bytes	1 MB + 32 KB + 320 bytes
IN currentFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
IN saturationLimit[]	64 bytes	64 bytes	64 bytes	64 bytes			
INOUT saturationFrame[N][N][BS][BS/32]	512 KB	8 KB	16 KB	32 KB			
3. subtractSuperBias		256 KB	512 KB	1 MB			
INOUT currentFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
INOUT biasFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
4. nonLinearityCorrectionPolynomial		128 KB	256 KB	512 KB			
INOUT currentFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
IN coeffOfNonLinearityPolynomial[][]	256 bytes	256 bytes	256 bytes	256 bytes			
5. subtractReferencePixelTopBottom					1MB (1 column composed of 8 blocks)		
INOUT currentFrame[N][N][BS][BS]	8 MB	1 MB					
6. subtractReferencePixelSides					8 MB (complete frame)		
INOUT currentFrame[N][N][BS][BS]	8 MB	8 MB					
7. detectCosmicRay		832 KB	1,62 MB	3,25 MB	832 KB	1,5 MB + 144 KB	3 MB + 288 KB
IN currentFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
IN sumXYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			
IN sumYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			
INOUT offsetCosmicFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
INOUT numberOfFramesAfterCosmicRay[N][N][BS][BS]	4 MB	64 KB	128 KB	256 KB			
8. progressiveLinearLeastSquaresFit		776 KB	1,5 MB	3 MB			
IN currentFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
INOUT sumXYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			
INOUT sumYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			
IN offsetCosmicFrame[N][N][BS][BS]	8 MB	128 KB	256 KB	512 KB			
IN saturationFrame[N][N][BS][BS/32]	512 KB	8 KB	16 KB	32 KB			
9. calculateFinalSignalFrame		512 KB	1 MB	2 MB			
IN sumXYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			
INOUT sumYFrame[N][N][BS][BS]	16 MB	256 KB	512 KB	1 MB			

Overall, assuming that the sensor frame is divided in 64 blocks, we can distinguish between four different execution phases executed one after the other and summarised as follows:

- During the *cluster phase 1,* 16 parallel executions on clusters are offloaded, each processing 4 blocks in parallel by *detect-Saturation subtractSuperBias* and *nonLinearityCorrection* functions.
- During the *cluster phase 2,* 8 parallel execution on clusters are offloaded, each processing 8 blocks (corresponding to 1 frame column) sequentially by the *subtractPixelTopBotton* function.
- During the *IO phase,* the complete frame is processed in a single IO core by the *subtractReferencePixelSides* function.
- During the *cluster Phase 3,* 64 executions on clusters are offloaded (being able to execute only 16 in parallel), each processing 1 block by *detectCosmicRay* and *linearLeastSquares-Fit* functions.

- Finally, once the stages within the loop finish, the *cluster phase* 4 offloads 64 executions on clusters (being able to execute only 16 in parallel), each processing 1 blocks by the *calculateFinalSignal-Frame* function.

4.2.3 Performance analysis: Average and Maximum Observed Execution Time

This section compares the *average execution time*, the *maximum observed execution time*, (also known as *high water-mark* in the critical real-time embedded domain) and the standard deviation of the infra-red application in milliseconds (ms), when parallelising the application using both, the native Kalray SDK, and the P-SOCRATES SDK developed within the project.

The execution time measurements presented in this section have been obtained by executing the application 100 times. We consider that the loop iterates 5 times (*#Groups = 5*). This section considers the observed execution time of the end-to-end execution of the application (labelled as *TOTAL*), and each of the execution phases within one loop iteration as presented in the previous section (labelled as *Cluster Phase 1, Cluster Phase 2, IO Phase, Cluster Phase 3* and *Cluster Phase 4*). The initialization and end phase of the application are not accounted for in this section.

Table 7 compares the average performance and the performance speed-up, considering as a baseline, the execution time of the sequential version of the application executed in one IO core. The table also presents the maximum theoretical speed-up that the MPPA can exhibit. This maximum theoretical speed-up corresponds to the number of cores that can potentially execute in parallel.

Table 7: Comparison of the average performance, in terms of ms and speed-up, of the infra-red application parallelised with the MPPA native and P-SOCRATES SDKs.

	Execution time (ms)			Speed-up		
	Sequential	MPPA Native SDK	P-SOCRATES SDK	MPPA Native SDK	P-SOCRATES SDK	Maximum theorical
TOTAL	**63140**	**8872,4**	**9093,2**	**7,1**	**6,9**	**256**
Cluster Phase 1	2710	162,64	150,34	16,7	18,0	48
Cluster Phase 2	562	147,56	120,08	3,8	4,7	8
IO Phase	612	612	612	1,0	1,0	1
Cluster Phase 3	8554	804,9	879,1	10,6	9,7	16
Cluster Phase 4	950	236,9	285,6	4,0	3,3	16

The infra-red application achieves similar performance speed-ups when parallelising and executing the application by using the Kalray SDK and the P-SOCRATES SDK, with 7.1x and 6.9x respectively. This speed-up however, is very far from the theoretical one (256x) due to the very limited parallelism exposed by the application in each of the execution phases.

When analysing each of the execution phases, speed-ups are similar for the Kalray and P-SOCRATES SDKs, with speed-ups of 16.7x and 18x during the first cluster phase, 3.8x and 4.7x during the second phase, 10.6x and 9.7x during the third phase and 4.0x and 3.3x during the fourth phase. The theoretical speed-up accounts for the maximum number of cores that can be used in each phase. Thus, in the first phase executions are spawned across the 16 clusters, each using up to 4 cores, and so 16*4 = 48. It is important to note that the speed-up also accounts for

the impact of transferring code and data at every offloading and the overhead of the OpenMP run-time, very similar in both the Kalray and P-SOCRATES SDKs.

Overall, we conclude that P-SOCRATES SDK does not degrade the average performance speed-up, being comparable to the MPPA native SDK.

Table 8 compares the maximum observed execution time (in ms) and the standard deviation of the different execution phases of the infra-red application when using the MPPA native SDK and the P-SOCRATES SDK.

With the P-SOCRATES SDK we observe smaller maximum execution times for *cluster phases 1* and *2* compared to the Kalray SDK. This is not the case for the *cluster phase 4,* in which the Kalray SDK obtains smaller execution times. *Cluster phase 3* obtains similar maximum execution times for the two SDKs. However, the standard deviation is smaller in the case of the P-SOCRATES SDK, revealing a smaller execution time variation than for the Kalray SDK. *Cluster phase 4* obtains a slightly higher standard deviation in the case of the P-SOCRATES SDK, although the value is already very small. The *IO phase* implements a sequential execution, thus exhibiting no variation in the execution time.

Table 8: Comparison of the infra-red application when parallelizing with the Kalray and the P-SOCRATES SDKs in terms of maximum observed execution time and standard deviation.

	Maximum Observed Execution time (ms)		Standard Deviation	
	MPPA Native SDK	P-SOCRATES SDK	MPPA Native SDK	P-SOCRATES SDK
Cluster Phase 1	210	160	27,28	1,81
Cluster Phase 2	180	130	4,18	0,89
IO Phase	612	612	0	0
Cluster Phase 3	1050	1110	318,78	277,51
Cluster Phase 4	240	290	4,65	4,99

4.2.4 Energy Consumption

Table 9 shows the average power (in W) and the energy consumption (in J) of the infra-red application when executing it sequentially (in one IO core) and in parallel, using the Kalray native SDK and the P-SOCRATES-SDK. The average power measured in both MPPA and P-SOCRATES SDK are very similar, resulting in similar energy consumptions. It is important to note that numbers presented in Table 9 have been obtained with the k1-power tool provided by Kalray, and so the execution time values differ from the ones obtained in Table 7, which are measured with instrumented code (with the k1-power tool the initialization and end phase of the application are also taken into account).

Table 9: Energy consumption of the infra-red case when parallelising with the Kalray and the P-SOCRATES SDKs.

	Sequential	MPPA Native SDK	P-SOCRATES SDK
Acquisition time (s)	75,84	30,38	24,81
Average power (W)	7,27	7,77	8,02
Energy (J)	551,14	235,93	199,12

4.2.5 Complexity of parallel code

Table 10 compares the source code of parallel version of the infra-red application by using the P-SOCRATES SDK (and so by applying OpenMP directives) and the Kalray native SDK to assess about the programming complexity.

There is an important increase in terms of the total number of lines of code and also in terms of the total number of function calls in the Kalray native SDK version with respect to the sequential version. This increment is due to the use of the MPPA API for process management and communication. The P-SOCRATES SDK instead, has a smaller impact on these metrics: the increment on the total number of lines of code with respect to the sequential version. This is due to the use of OpenMP pragmas and the control flow and data structures related with them [9].

Notice also the increment on the number of source code files needed for the MPPA SDK with respect to the sequential and the P-SOCRATES SDK. The reason is because the user has to manually separate the code to be executed in the IO and the code to be executed in the cluster cores in order to compile them separately and so create different binaries for each of the phases. The P-SOCRATES SDK instead automatically performs the compilation process as presented in deliverable D2.2.2 [7].

Table 10: Complexity on the parallel programming when using the Kalray and the P-SOCRATES SDKs.

	Sequential	MPPA native SDK	P-SOCRATES SDK
Total number of lines	885	1895	1230
Total number of used functions	23	282	50
Total source-code files	4	8	4
Extra functionality	-	-	- OpenMP untied task model - Fine grain synchronization dependencies

[9] It is worth noting that the increment on the number of function calls in the P-SOCRATES SDK is also due to an inefficient use of the array sections in the OpenMP pragmas, as arrays must be contained in consecutively memory positions. Therefore, each of the blocks of the arrays has to be passed as an independent variable and thus, a function call is needed for each of the variables. The ideal solution would be to implement a unique function call inside a loop indexing the specific block of the array. This remains as a future work.

4.3 Parallel Complex Event Processing Engine

4.3.1 Evolution of the pCEP

A Complex Event Processing (CEP) engine can be briefly described as a software solution to collect raw data streams, process them, and derive meaningful value-added information to a third party. In the early stages of P-SOCRATES project a CEP engine was available thanks to the outcomes of the FIWARE project, where the SOL/CEP engine was developed. Since then, the Internet of Everything Lab from Atos Research & Innovation division has taken the lead of the development roadmap of the CEP Engine as a whole, not only for P-SOCRATES project but as a more ambitious plan to enhance the tool and provide a CEP-as-a-Service solution into its catalogue of services.

During the development lifecycle of the engine a complete new approach was pursued. The engine has been unbundled in a set of modules which communicate between them with specific interfaces. This new release has been called µCEP since it is a lightweight version compared to the previous one, where neither additional software dependencies nor a Java Runtime Environment are needed. This version is implemented in C++ and compiled to run in common x86 machines or even mini computers such us the ARM-powered RaspberryPi. With respect to the parallelization of the µCEP engine, a new fork was created to open a new development branch from the master repository, with the goal to apply parallelization strategies to the implementation (Figure 19).

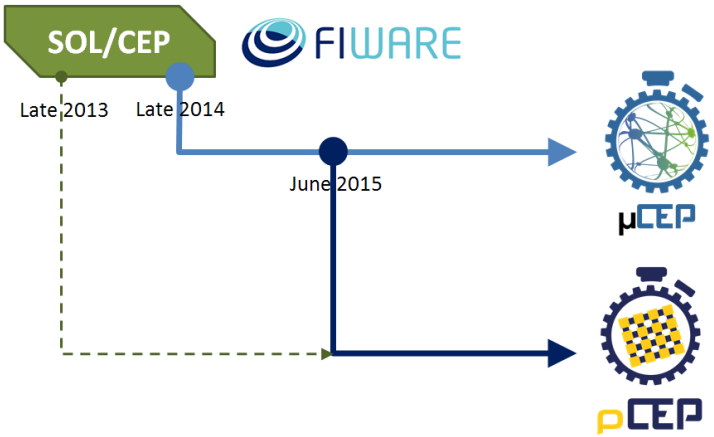


Figure 19: CEP Engine development roadmap.

The very initial release of the pCEP (parallel Complex Event Processing) consisted in a yet-sequential version of the former µCEP but adapted to the Kalray MPPA board architecture, splitting the master project in sub-projects, each of them to be compiled and run in the workstation and the IO-core portion of the MPPA. Since then, subsequent improvements were done in order to go parallel, extracting part of the code from the IO-core and porting it to the compute-cluster portion of the MPPA. After that, the remaining workstation source code was incorporated into the IO-core so not to have any external code running outside the MPPA.

4.3.2 Parallelization Strategy on the MPPA

The architecture of the pCEP is composed by three main elements. The *Event Collector* module responsible for acquiring data from external sources and converting them into Events; the output is the responsibility of the *Complex Event Publisher* module. The core of the solution is the *Complex Event Processor* module, which entails the case study of the work. In brief, as depicted in Figure 20, this module receives *Events* that trigger *rules* (conditions) and output *Complex Events*, both being previously defined in a DOLCE file (see deliverable D1.3 "Low-latency complex event processor" [8]). Those rules invoking *complex functions* make use of the *Instruction Evaluator* sub-module, which is the part of the pCEP with the most demanding and time consuming features. Particularly, the *Complex Event Processor* module has been rewritten to offload to the accelerator devices the catalogue of *complex functions*.

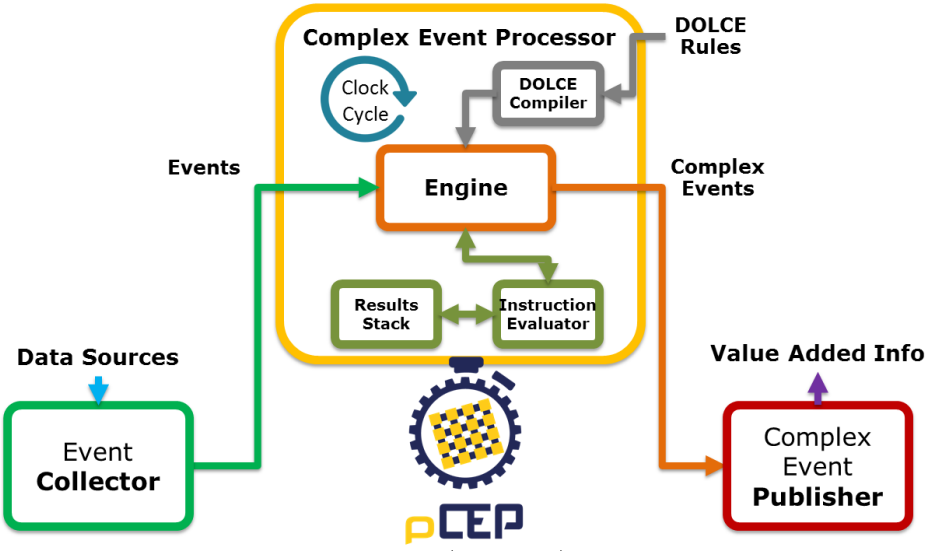


Figure 20: Complex Event Process module of the pCEP.

In the first place, we ran an experiment to understand the performance of the initial catalogue of *complex functions* (composed of sum(), average(), count() and diff() functions) and conclude that they are not algorithmically complex enough so as to benefit from being executed in parallel. Table 11 collects some measurements extracted from the experiments carried on the average() function.

Table 11: Speed-Up sequential vs. Kalray vs. P-SOCRATES.

		Total time (s)			Speed-Up	
		Sequential	Kalray SDK	P-SOC SDK	Kalray SDK	P-SOC SDK
avg() function with 50 events entry	1 event window	33,81	90,04	41,24	0,375	0,819
	2 event window	66,23	190,63	80,88	0,347	0,818
	4 event window	127,06	376,40	155,15	0,337	0,818
	…	…	…	…	…	…

As shown in the table, there is a clear underperformance when going parallel (more in the native Kalray SDK) because the average() function executes so fast that the overhead introduced by the parallelization levels induce a huge time penalty in the performance. As in the first use case presented this is to be expected due to the overhead penalty with very small parallel code.

Moreover, regardless of the number of events processed (so the length of the event pool), the effect remains the same and the speed-up gain (in this case, reduction) remains the same. Therefore, as explained in details in section 6.2 of Deliverable 3.4 [9], another complex function was added to the pCEP engine, the Fast Fourier Transform (FFT) algorithm to compute the Discrete Fourier Transform (DFT). With this, the pCEP engine allows testing the P-SOCRATES SDK, while also benefits from an extended catalogue of *complex functions*, enabling its usage in a wider set of application scenarios.

4.3.3 Complexity of parallel code

The pCEP engine evolves from an existing application, whose source code is written mostly in C++, with around 13.000 lines of code split in 120 files. Because of using an Object-oriented programming (OOP) paradigm, and due to the dynamic nature of a stream analytics tool, specific modifications were included in the application to make it compatible with the hardware architecture and the tools used in the project.

On one hand, given that the current P-SOCRATES compiler could only introduce annotations in C code, we identified the specific portions of code that had to be offloaded in the accelerator, and thus manipulated by the compiler, and rewrite them in C code. On the other hand, as in previous use cases, the dynamic nature of the pCEP engine clashed against the limitations in memory allocation at the accelerator area of the Kalray MPPA. Therefore, it was advisable to control the limits in memory consumption of the *events*, *event pools* and *complex events*, refactoring dynamic lists composed by structures into static arrays.

Additionally, a key step occurred during the process of dividing the code into files that were to be compiled for the IO core, and those to be compiled for the compute clusters. Despite the previous limitations, the P-SOCRATES SDK empowers a huge advancement, as it allows application developers to manage a unique project regardless of the underlying compilation procedures, avoiding the maintenance of several separated portions of the whole application. Figure 21 shows the submodules that are run by each core types of the MPPA, while in Table 12 we depict the files and number of lines of code and compare from its implementation while using the Kalray SDK and P-SOCRATES SDK, respectively. Even if the numbers do not vary a lot, the developer clearly benefits from the ability to maintain the code in a single repository.

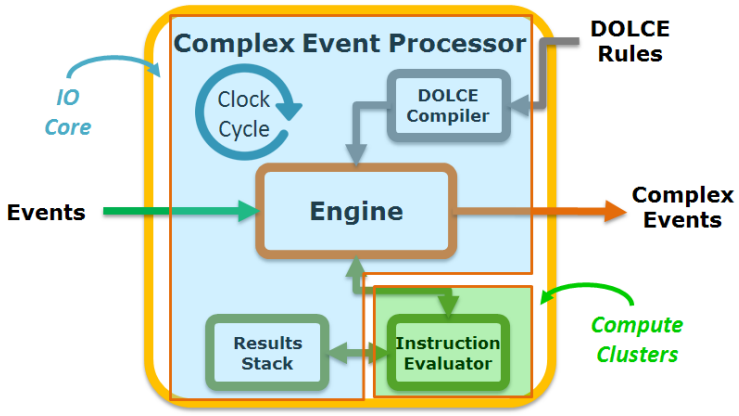


Figure 21: Division of pCEP submodules per compilation scheme.

Table 12: Complexity of pCEP code.

	Files		Code	
	Kalray SDK	P-Socrates SDK	Kalray SDK	P-Socrates SDK
C++	34	31	9981	8753
C/C++ Header	80	74	2945	2722
C	0	4	0	753
yacc	1	1	651	651
lex	1	1	251	251
make	1	1	101	101
XML	1	1	36	36
Bourne Shell	1	1	21	21
TOTAL	**119**	**114**	**13986**	**13288**

4.3.4 Performance analysis: Average and Maximum Observed Execution Time

Following the same procedure as for the previous use cases, in this section we compare the *average execution time*, the *maximum observed execution time*, and the standard deviation of the pCEP engine when it is executed with a list of *events* that trigger *rules* invoking the execution of *complex functions*. This implies spawning to compute clusters and offloading parallel executions of the *complex functions* on the accelerator devices. Sequential vs. native Kalray SDK vs. P-SOCRATES SDK versions are compared.

In order to obtain relevant results in terms of execution time we performed a set of tests with intensive calculations inside the compute clusters by invoking high computational demanding complex functions such as the proposed FFT. In this case, for each input event we launched the execution of a set of 256 FFTs of 512 elements. Since the usage of different number of threads changes the performance, also the experiment cases using from 1 up to 256 tasks have been evaluated. Results are shown in Table 13, where performance improvement (speed-up against sequential) is achieved in most of the cases.

Table 13: pCEP average execution time and Speed-Up for various cases.

	Execution time (secs.)			Speed-Up	
Tasks	Sequential	Kalray SDK	P-Socrates SDK	Kalray SDK	P-Socrates SDK
	135,65				
1		152,77	131,45	0,89	1,03
2		76.28	65,73	1,78	2,06
4		38,01	32,82	3,57	4,13
8		18.99	16,41	7,14	8,27
16		9,53	8,29	14,24	16,37
32		9,53	8,29	14,24	16,36
64		9,53	8,29	14,24	16,37
128		9,53	8,28	14,24	16,37
256		9,53	8,29	14,24	16,37

A detailed view of the results is showed in Figure 22. The speed-up increases when the number of tasks grows, achieving the best performance (speed-up x16) when using 16 tasks or more, since all the threads available in the clusters are used. The behaviour of the pCEP engine implemented with the P-SOCRATES SDK offers better speed-ups than the Kalray native implementation.

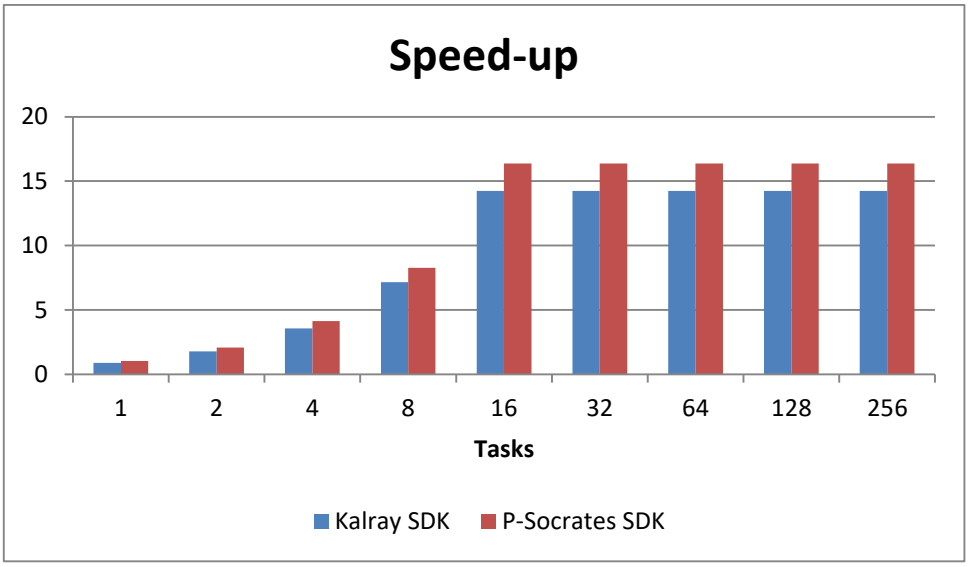


Figure 22: pCEP performance when using Kalray SDK and P-Socrates SDK vs. sequential.

By running the application a hundred of times we obtain the worst-case performance for each test and the standard deviation against the average-case performance. In Table 14 we depict the results for the cases of 8, 16 and 32 tasks. We can see that in this case, the P-SOCRATES SDK has a higher standard deviation when all cores are occupied. However, in both cases, there is a very high variability.

Table 14: Average, maximum execution times and standard deviation.

Tasks	Average		Worst case		Standard Deviation	
	Kalray SDK	P-SOC SDK	Kalray SDK	P-SOC SDK	Kalray SDK	P-SOC SDK
8	18,991	16,410	19,056	16,440	16,22	12,89
16	9,526	8,285	9,537	8,300	4,45	5,60
32	9,526	8,289	9,549	8,310	5,36	7,30

4.3.5 Energy Consumption

Following the same methodology as introduced in Section 4.1.7, we obtain the power and energy consumption of the accelerator using the 'k1-power' tool provided by Kalray. The chart in Figure 23 shows how the power consumption remains quite constant for all the analysed cases, with values around 7 to 8 Watts. Because the execution time decreases with the number of tasks used, especially with the P-SOCRATES implementation, it is in those cases that the energy consumption is significantly slower (see Figure 24). No big differences between both parallelized flavours were observed.

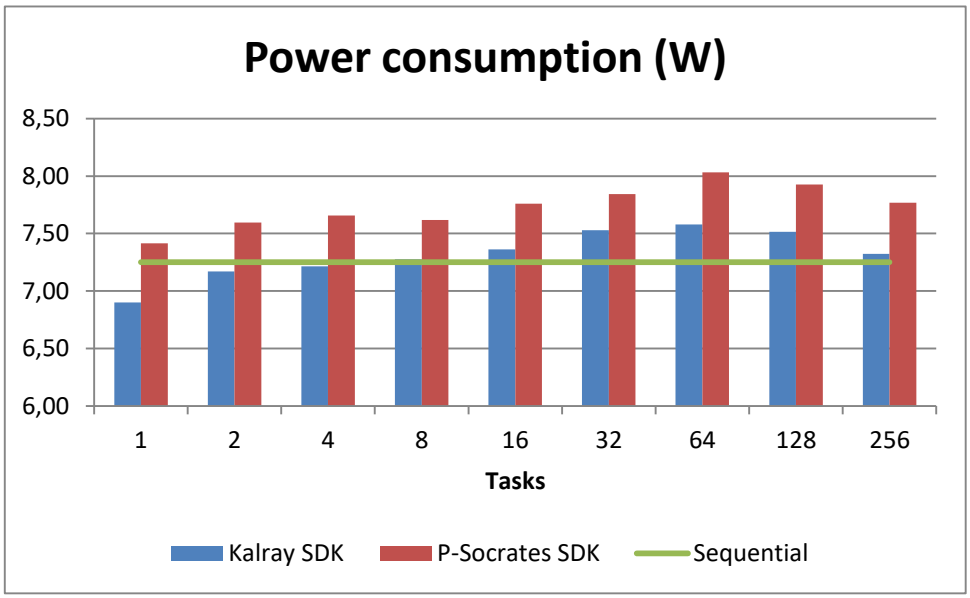


Figure 23: Power consumption of the pCEP application.

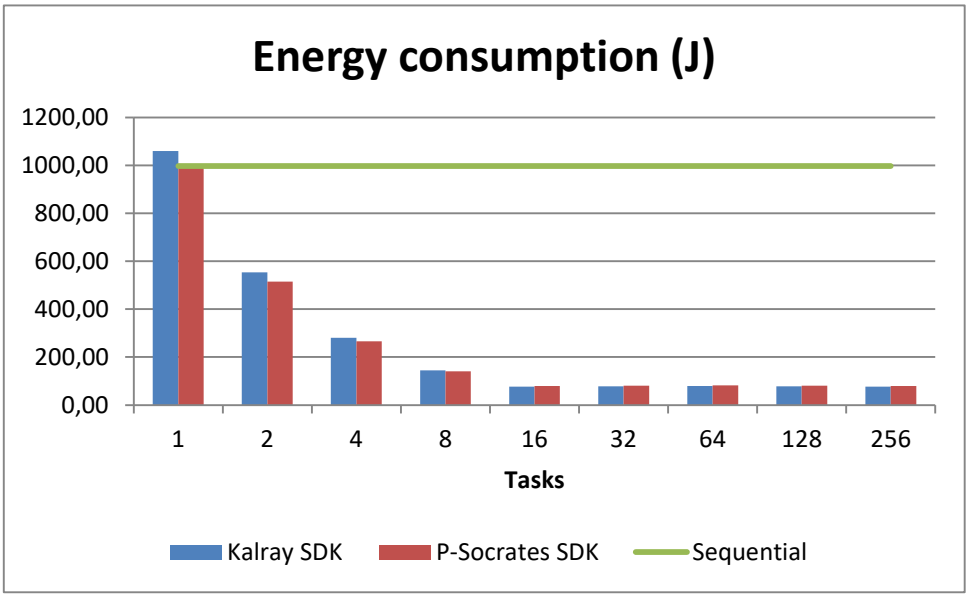


Figure 24: Energy consumption of the pCEP application.

5 Conclusions

This deliverable presented the result of the evaluation process of the P-SOCRATES SDK with the three use cases considered in the project. The delivered presented the parallelization approaches used, as well as the comparison of the performance (average and maximum observed values), complexity of coding and power consumption between a sequential version, a version parallelized with the native SDK provided by Kalray, and finally the new SDK implemented by P-SOCRATES (using dynamic scheduling approaches).

The results clearly show that the P-SOCRATES SDK provides no performance and power penalty (and even improvement in some cases) compared to the native SDK, whilst providing more functionality and less code complexity, validating the respective goals of the project.

References

[1] P-SOCRATES Deliverable 1.2.2. Preliminary evaluation report. Delivery date: 31 March 2016.

[2] P-SOCRATES Deliverable 4.3.2. Overall schedulability analysis. Delivery date: 31 March 2016.

[3] P-SOCRATES Deliverable 4.4. Overall analysis of the use-cases. Delivery date: 31 December 2016.

[4] P-SOCRATES Deliverable 6.4. SW optimizations and applicability to other many-core processors. Delivery date: 31 March 2016.

[5] P-SOCRATES Deliverable D4.3.1. Independent Schedulability Analysis. Delivery date: 31 March 2015.

[6] P-SOCRATES Deliverable 1.2.1. Use cases for preliminary evaluation. Delivery date: 31 March 2015.

[7] P-SOCRATES Deliverable 2.2.2. Automatic extraction of parallelism. Delivery date: 31 March 2016.

[8] P-SOCRATES Deliverable 1.3. Complex Event Processor. Delivery date: 31 March 2016.

[9] P-SOCRATES Deliverable 3.4. Integrated mapping/scheduling infrastructure for the Use Cases. Delivery date: 31 December 2016.