Final Report

Parallel Programming Models for Space Systems

(ESA Contract No. 4000114391/15/NL/Cbi/GM)

Main contractor	Subcontractor
Barcelona Supercomputing Center (BSC) Eduardo Quiñones, eduardo.quinones@bsc.es	Evidence Srl. Paolo Gai, pj@evidence.eu.com

Dissemination level: All information contained in this document is public

June 2016

Table of Contents

1 Introduction

High-performance parallel architectures are becoming a reality in the critical real-time embedded systems in general, and in the space domain in particular. This is the case of the NGMP GR740 featuring four LEON4FT SparcV8 cores with fault tolerance support, suitable for the space domain. In order to exploit the performance opportunities of these processors, the use of advance parallel programming models becomes of paramount importance.

This project evaluates the use of OpenMP parallel programming model in the space domain. OpenMP is a well-known and widely adopted parallel programming model in the high-performance domain, that incorporates a powerful tasking and acceleration execution models to easily express very sophisticated types of dynamic, fine-grained and irregular parallelism, facilitating to exploit the huge performance opportunities of the newest multi-core many-core heterogeneous embedded systems.

OpenMP addresses key issues in practical programmability of current real-time (RT) systems: the mature support of task definition augmented with features to express data dependencies among them is particularly relevant for RT systems, in which applications are typically modelled as periodic direct acyclic graphs (DAG).

Moreover, OpenMP is supported by a large set of current multi-core and many-core architectures, including the NGMP GR740. Developing systems that can be easily ported among different platforms is of paramount importance to exploit the performance opportunities brought by hardware evolutions.

This final report includes deliverables D1.1 (Annex I), D2.1 (Annex II) and D3.1 (Annex III), in which OpenMP is evaluated considering four key performance parameters: programmability, performance, time predictability and OS services required. Concretely:

- In Annex I, Deliverable *D1.1 - Report on parallelisation experiences for the space application* investigates OpenMP from a: (1) *programmability* point of view, devising three parallelisation strategies for a space application, a pre-processing sampling application for infrared H2RG detectors targeting both, homogeneous and heterogeneous parallel architectures; (2) *performance* point of view, evaluating the speed-up on four different parallel architectures: A 16-core Intel Xeon, a 274-core MPPA; and (3) *time predictability* point of view, evaluating the worst-case response time of the OpenMP tasking model consider both, dynamic and static allocation approaches.
- In Annex II, Deliverable *D2.1 - Report on the evaluation of current implementations of OpenMP* investigates the performance and memory overhead introduced by the OpenMP run-time when increasing the number of parallel tasks, and proposes a set of OS services to execute of OpenMP program with a minimum code footprint impact and memory requirements for OS targeting the highest critical systems,
- In Annex II, Deliverable *D3.1 - Report on the applicability of OpenMP4 on multi-core space platforms* evaluates the parallel version of the space application considering in this project under two parallel platforms suitable for space domain, including the RTEMS SMP OS and two LEON-based processors, i.e. a 4-core LEON3 implemented on a FPGA and a 4-core NGMP-GR740.

Overall, we conclude that OpenMP is a very convenient parallel programming model for real-time embedded systems in general, and space systems in particular.

This project has successfully achieved all project objectives, going even beyond of what was stated in the *Statement of Work* document, by evaluating OpenMP on a parallel platform relevant for space and so reaching a TRL of 4.

2 Future work: Next development activities to reach higher TRL (5/6)

The next development activities needed to reach TRL 5 or 6 and so consolidate the adoption of OpenMP into the space domain are the following:

1. This project demonstrated that the overhead introduced by the run-times may diminish the benefit brought by parallel computation. Therefore, the development of efficient and lightweight OpenMP run-times with smaller overheads is desirable to further exploit finer-grain parallelism and so increase the performance of multi-core architectures such as the NGMP GR740.
2. Despite this project has proved that OpenMP is time predictable, the techniques used to characterise the timing behaviour of parallel regions (tasks) were not accurate enough to be used in an industrial environment, e.g. the overhead introduced by run-time and OS context switch were not considered. Therefore, further investigations on sound and trustworthy timing analysis methods are required.
3. This project proposed a set of services required to execute OpenMP run-time on top of Erika Enterprise OS suitable for critical real-time embedded systems. Its implementation however is still required.
4. Finally, this project has not addressed the functional correctness of OpenMP programs required to execute on safety critical environments such as space. Investigations on compiler techniques to ensure functional correctness are therefore needed.

Annex I - D1.1. Report on parallelisation experiences for the space application

Eduardo Quinones, Maria A. Serrano
{eduardo.quinones@bsc.es, maria.serrano@bsc.es}

June 2016

Dissemination level: All information contained in this document is public

Parallel Programming Models for Space Systems

(ESA Contract No. 4000114391/15/NL/Cbi/GM)

Abstract

This report briefly summarises the OpenMP tasking and acceleration execution models, and devises three parallelisation strategies of the pre-processing sampling application for infra-red H2RG detectors using the two OpenMP execution models, and targeting shared and a distributed memory architectures. The presented parallelisation strategies are then evaluated considering programmability, performance speed-up and time predictability.

Table of Contents

1 Introduction

Multicores and many-cores are becoming a reality in the critical real-time embedded systems in general, and in the space domain in particular. Examples include the NGMP GR740 quad core LEON and 256 MPPA Kalray. In order to exploit the performance opportunities of these processors, the use of advance parallel programming models becomes of paramount importance.

OpenMP is a well-known and widely adopted parallel programming model in the general-purpose and high-performance domains. Originally focused on massively data-parallel, loop-intensive applications, the latest specification of OpenMP (version 4.5, released on November 2015) has evolved to consider very sophisticated types of dynamic, fine-grained and irregular parallelism. It also incorporates new features to exploit the performance of the newest many-core heterogeneous embedded systems, allowing to couple a main *host* processor to one or more acceleration devices in which highly parallel code kernels can be offloaded for improved performance/watt.

OpenMP addresses key issues in practical programmability of current real-time (RT) systems: It provides mature support for highly dynamic task parallelism, augmented with features to express data dependencies among tasks. Such points make OpenMP particularly relevant for RT systems, in which applications are typically modelled as periodic direct acyclic graphs (DAG). Overall, OpenMP results in an excellent choice for current and future RT systems as:

1. it provides the abstraction level required to program parallel applications, while hiding the complexities of parallel architectures, and
2. it provides the necessary methodology to exploit the huge performance opportunities of the newest many-core processors, facilitating the migration of RT systems from multicore to many-core platforms.

Developing systems that can be easily ported among different platforms is of paramount importance to exploit the performance opportunities brought by hardware evolutions. This is in fact one of the objectives of parallel programming models.

Unfortunately, OpenMP adopts a parallel execution model that differs in many aspects from the RT execution model: The programming interface is completely agnostic to any timing requirement that the target application may have.

This document evaluates the use of OpenMP in the space domain by presenting our experiences of parallelising a space application, i.e. a pre-processing sampling application for infrared H2RG detectors. Concretely, we present three different parallelisation strategies targeting multi-core and many-core processor architectures. These parallelisation strategies are then evaluated considering three key evaluation parameters: programmability, performance speedup and time predictability.

The rest of the document is organized as follows. Section 2 describes the space application considered in this project and Section 3 introduces the two execution models of OpenMP: the *tasking* and the *acceleration* models. Sections 4 and 5 presents the experiences of parallelising the space application targeting a shared memory architecture, and evaluate the parallel versions from a programmability and performance speed-up point of view. Section 6 evaluates the same parallel versions but from a time predictability point of view. Section 7 presents the experiences of parallelising the space application targeting a many-core heterogeneous architecture. Finally, conclusions are presented in Section 8.

2 Application description

This project considers a pre-processing sampling application for infrared H2RG detectors composed of nine sequential stages (in parenthesis, we specify the name of the C function implementing the corresponding stage):

1. *Get frame* stage (named *getCoAddedFrame*). It simulates the acquisition of a given number of readouts from the H2RG sensor frame (the number of readouts is defined by the *#frame* parameter), and copies it into a 2048x2048 array structure.
2. *Saturation detection* stage (named *detectSaturation*). It detects when pixels go into saturation in order to reduce the readout noise.
3. *Super-bias subtraction* stage (named *subtractSuperBias*). It removes pixel-to-pixel variation by subtracting a bias frame from the detector's frame.
4. *Non-linearity correction* stage (named *nonLinearityCorrection*). It corrects the frame using a 4th order polynomial.
5. *Phase 1 - reference pixel subtraction* stage (named *subtractPixelTopBottom*). It 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
6. *Phase 2 - reference pixel subtraction* stage (named *subtractPixelSides*). It removes common noise by computing the average of lateral pixels.
7. *Cosmic ray detection* stage (named *detectCosmicRay*). It estimates the disturbances by cosmic ray.
8. *Linear least square fit* stage (named *linearLeastSquaresFit*). It detects disturbances by cosmic ray.
9. *Final signal frame* stage (named *calculateFinalSignalFrame*). It updates the frame in the required format.

Stages 1 to 8 are executed within a loop iterating given number of times (bounded by the *#Groups* parameter). *Figure **1*** shows the sequential execution of the nine stages defined above.

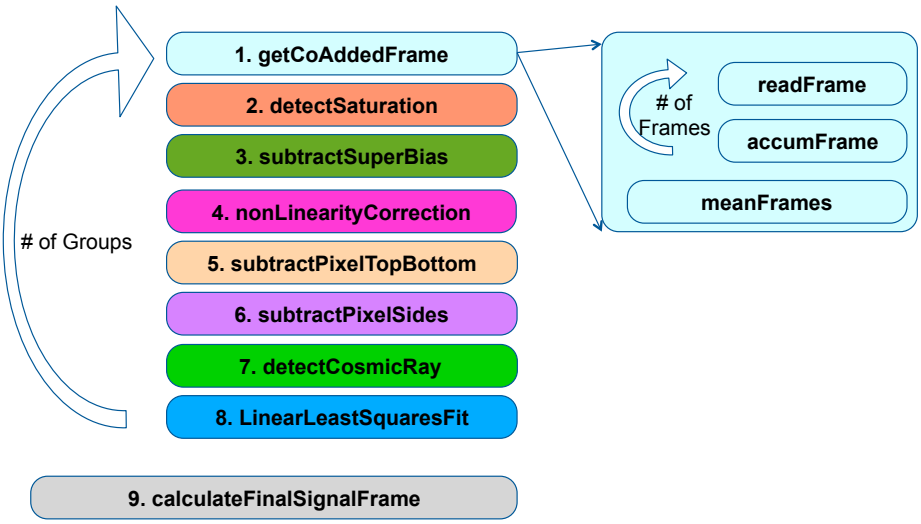

Figure 1. Pre-processing sampling application stages.

2.1 Sensor frame readout simulation

The *getCoAddedFrame* 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. Because the variations on the input data introduced by the random system call does not impact on the overall execution time of the application, the use of random values is not representative of the physical data acquisition occurring into a real system.

To that end, the results presented in this project do not consider this first stage, and assume that the physical process of sensor data acquisition is overlapped with the computation of the previous acquired frame as shown in *Figure **2***. In any case, the overall performance of the application will be limited by the readout rate of the H2RG sensor[1].

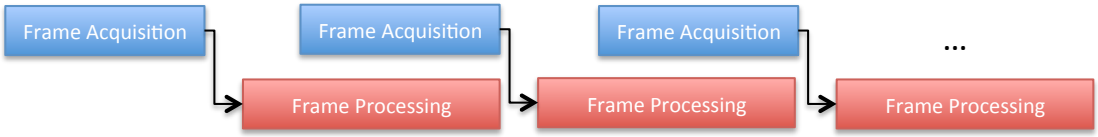

Figure 2. Pipeline parallelisation strategy between the frame acquisition and frame processing.

3 The OpenMP parallel programming model

OpenMP[2] (Open Multi-Processing) is a de-facto parallel programming standard for shared memory architectures widely adopted in high performance computing domain. With the recent introduction of many-core heterogeneous processors targeting the embedded domain, OpenMP is increasingly receiving a lot of attention from this domain as well.

3.1 Evolution of OpenMP

Up to specification 2.5[3] (2005), OpenMP supported the traditional *fork-join execution model* focused on massively data-parallel, loop-intensive applications, following the *single program multiple data* programming paradigm. To do so, OpenMP provided *threads* as an abstraction layer upon which programmers could assign the code segment to be executed in parallel. Latter, during the parallel execution, the OpenMP run-time assigns threads to cores following the execution model defined by OpenMP specification.

With the introduction of OpenMP 3.0 specification[4] (2008), the `task` directive was introduced, exposing a higher level of abstraction to programmers. A task is an independent parallel unit of work, which defines an instance of code and its data environment executed by a given available OpenMP thread. This new model, known as *tasking model*, provides a very convenient abstraction of parallelism, being the run-time in charge of scheduling tasks to threads.

With the introduction of OpenMP 4.0 specification[5] (2013), OpenMP evolved to consider very sophisticated types of fine-grained, irregular and highly unstructured parallelism, by enabling a mature support to express dependencies among tasks. Moreover, it incorporated for the first time, a new *acceleration model* including features for off-loading computation and data transfers between the host and the accelerator device.

[1] Full-frame readout 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
[2] http://openmp.org
[3] http://www.openmp.org/mp-documents/spec25.pdf
[4] http://www.openmp.org/mp-documents/spec30.pdf
[5] http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

The latest OpenMP specification, 4.5[6] (2015), enhances the previous accelerator model by coupling it with the tasking model.

3.2 Tasking and acceleration model of OpenMP4.5

An OpenMP program starts with a single thread of execution, called the *master* or *initial* OpenMP thread, that runs sequentially. When the thread encounters a `parallel` construct, it creates a new *team* of threads, composed of itself and *n − 1* additional threads (*n* being specified with the `num_threads` clause). The use of *work-sharing* constructs allows specifying how the computation within a parallel region is partitioned among the threads.

3.2.1 Tasking model

The *tasking execution* is supported by the `task` directive. When a thread encounters it, a new task region is generated from the code contained within the task. The execution of the new task region can then be assigned to one of the threads in the current team for immediate or deferred execution, based on the `depend` clause[7], which allows to describe a list of `in`, `out` or `inout` dependences on target data items. If a task has an `in` dependence on a variable, it cannot start executing until the set of tasks that have `out` and/or `inout` dependences on the same variable complete. Dependences can only be defined among sibling tasks, i.e. tasks created within the same task region.

All tasks bound to a given parallel region are guaranteed to have completed at the implicit barrier at the end of the parallel region, as well as at any other explicit `barrier` directive. Synchronization over a subset of explicit tasks can be specified with the `taskwait` directive, which forces the encountering task to wait for all its first-level descendants to complete before proceeding.

3.2.2 Accelerator model

The OpenMP *accelerator model,* supported by the `target` directive, implements a host-centric model in which the host is the responsible of orchestrating the offloads to the accelerator devices. When a `target` directive is encountered, a new *target task* is generated, enclosing the target region. The target task is completed after the execution of the target region finishes. The target task is executed immediately by the encountering thread unless a `nowait` clause is present, which may defer the execution of the *target task*.

The enclosed target region is sequentially executed in the accelerator device by the initial thread until a parallel directive is encountered. The source-code implementation enclosed within a `target` must ensure that the target region executes as if it were executed in the data environment of the accelerator device[8]. The `map` clause specifies the data items that will be mapped into current target's data environment, copying the data value either from the host *to* the device (using the `to` clause), or *from* the device to the host (using the `from` clause).

One of the most interesting features of the new accelerator model supported in v4.5 specification is the integration of the acceleration and the tasking model. The `target` directive supports the `depend` clause, which allows describing `in`, `out` or `inout` dependences on data items among tasks.

[6] http://www.openmp.org/mp-documents/openmp-4.5.pdf

[7] OpenMP also includes the `untied`, `if` and `final` clauses that are not explained in this report.

[8] Unless an `if` clause is present and its associated expression evaluates to *false*.

4 Parallelisation strategy 1

This project considers a *data parallelisation strategy* in which the frame is divided into *NxN* blocks, each being potentially executed in parallel by application stages.

Unfortunately, there exist data dependencies among the different stages, which limit application's parallelization degree. Concretely, the two phases of the *reference pixel subtraction* stage process the elements of the frame in an order different from other stages, requiring previous stages to complete before starting them.

*Figure **3*** shows the data dependencies, in the form of a *task dependency graph* (TDG)[9], existing among the different stages when dividing the frame in 16 blocks (N = 4) and assuming *#Groups = 1*. As shown, functions *detect-Saturation*, *subtractSuperBias*, *nonLinearityCorrection*, *detectCosmicRay*, *linearLeastSquares-Fit* and *calculateFinalSignal-Frame* can process the 16 blocks in parallel. This is not the case of *subtractPixelTopBottom*, and *subtractPixelSides* functions, which reduce considerably the level of parallelism. The Figure also includes the *getCoAddedFrame* function that cannot be parallelised as explained above.

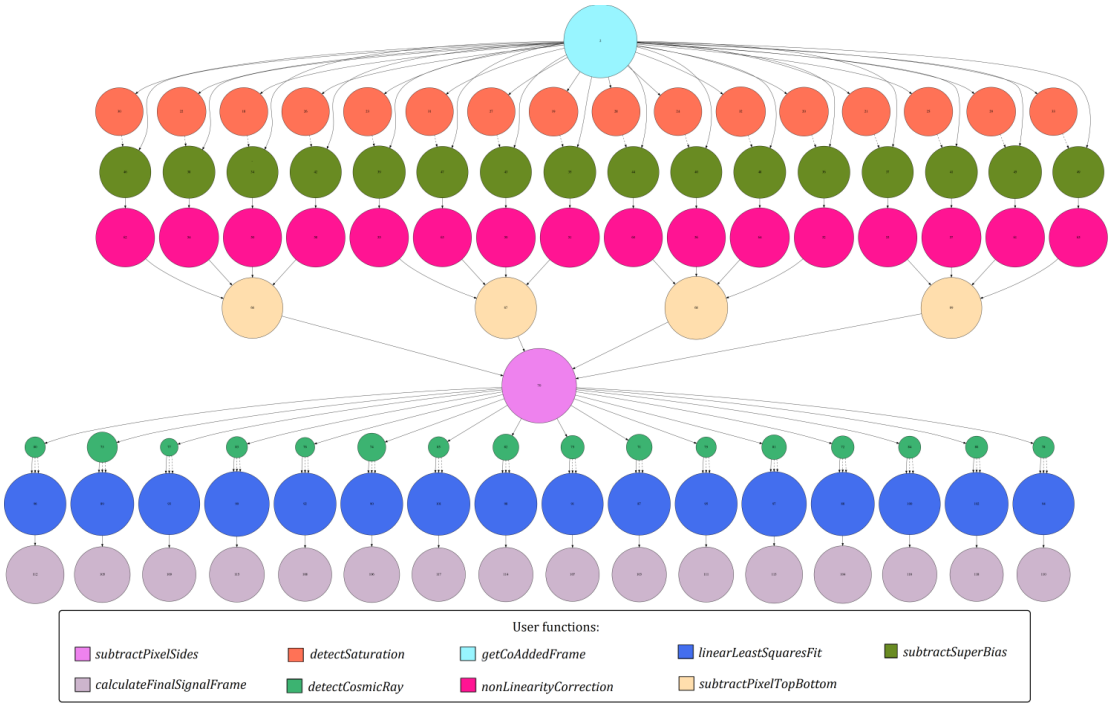

Figure 3. Data dependencies (in the form of TDG) among stages when dividing the frame in 16 blocks.

4.1 Parallel implementation with the tasking model

Figure **4** shows the use of the `task` directive, on a portion of the space application; concretely when the functions *subtractSuperBias, nonLinearityCorrectionPolynomial, subtractReferencePixelTopBottom* and *subtractRefe-rencePixelSides* are called.

For each of the blocks in which the frame array data structure is divided (determined by the `DIM_Y` and `DIM_X` parameters in the source code), the for-loop processes each block with the corresponding application stage. The computation of each stage is assigned to a task,

[9] The TDG has been generated with Mercurium (https://pm.bsc.es/mcxx) , source-to-source compiler developed at BSC.

enriched with the `depend` clause to express the dependencies existing among the different stages and shown in *Figure **3***. Hence, when the computation of function *subtractSuperBias* upon block [i,j] finishes, the computation of function *nonLinearityCorrectionPolynomial* upon block [i,j] can start executing as defined by the input/output dependency over `p[0:bs]`.

When dependencies cannot be defined, the use of a `taskwait` directive acts as a barrier, ensuring that all tasks complete before continuing the computation. A `taskwait` directive is used to synchronise the execution of *subtractReferencePixelTopBottom* and *subtractReferencePixelSides* functions.

```
...
for (i=0; i < DIM_Y; i++)
  for (j=0; j < DIM_X; j++)
    p_block_type p = (*currentFrame[groupNumber-1])[i][j];
    #pragma omp task depend (in: biasFrame[i][j], inout: p[0:bs])
                     firstprivate(i,j)
      subtractSuperBias(*currentFrame[groupNumber-1],biasFrame,i,j);

for (i=0; i < DIM_Y; i++)
  for (j=0; j < DIM_X; j++)
    p_block_type p = (*currentFrame[groupNumber-1])[i][j];
    #pragma omp task depend (in: coeffOfNonLinearityPolynomial)
                     depend (inout: p[0:bs]) firstprivate(i,j)
      nonLinearityCorrectionPolynomial(*currentFrame[groupNumber-1],
                                       coeffOfNonLinearityPolynomial,
                                       i, j);
#pragma omp taskwait
for (j=0; j < DIM_X; j++)
  #pragma omp task firstprivate(j)
    subtractReferencePixelTopBottom(*currentFrame[groupNumber-1], j);

#pragma omp taskwait
subtractReferencePixelSides(*currentFrame[groupNumber-1]);
...
```

Figure 4. *Portion of the application parallelised with the tasking model.*

4.2 Performance evaluation

*Figure **5*** shows the performance speed-up resultant of the parallelisation, when varying the number of blocks in which the sensor frame is divided, ranging from 1 to 1024. Experiments have been conducted on a two Intel(R) Xeon(R) CPU E5-2670 processor, featuring 8 cores each and 20 MB L3. The performance speed-up has been computed taking as a baseline the sequential execution of the application, i.e. when the number of blocks equals to 1.

As expected, the performance speed-up increases as the number of blocks in which the frame is divided increases as well, reaching a performance peak of 7x when the frame contains 64 blocks. Then, the performance starts degrading when the sensor frame is divided in more than 64 blocks, having almost no performance benefit when considering 1024 blocks.

This performance degradation is because of a twofold reason:

1. As the number of blocks in which the sensor frame is divided increase, the number of tasks created at run-time increase as well, which in turns increases the run-time overhead.

2. The size of the block computed by each task is too little, and so the run-time overhead introduce by the run-time dominates on the execution.

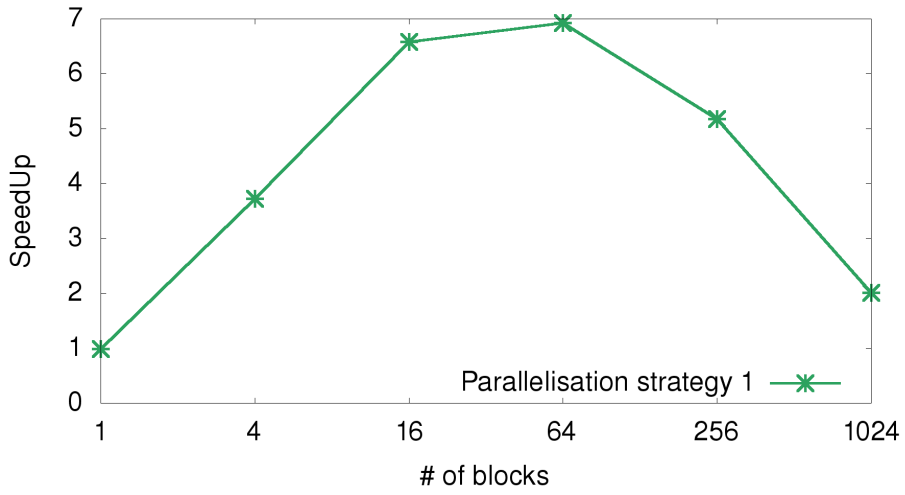

Figure 5. Application's performance speed-up when parallelizing considering the tasking and fork-join execution models.

*Table **5*** relates the number of blocks in which the sensor frame is divided, with the number of tasks created[10] and the size of each block processed by tasks. Hence, when matrix is divided in 1024 blocks, 26784 tasks are created, each processing a matrix of 64x64 elements. The workload to be computed by each task is too little (and so its execution time), making run-time overhead resultant of managing such a high number of tasks, to dominate over application's execution time. Deliverable *D2.1 - Report on the evaluation of current implementations of OpenMP4*, further discusses it.

Table 1. Number of created tasks at run-time and size of the sub-frame processed by each task, when dividing the sensor frame in blocks.

Number of blocks	Created tasks	Block size
1	31	2048x2048
4	114	1024x1024
16	436	512x512
64	1704	256x256
256	6736	128x128
1024	26784	64x64

4.3 Analysis of the parallel execution

In order to further investigate the parallel performance of the space application, we use Paraver[11], a performance analysis tool developed at BSC, used to visualize how parallel applications make use of computing resources. Paraver is a flexible data browser that enables the support of new parallel platforms (e.g. the multi-core NGMP GR740) by simply capturing the events under analysis (e.g. execution time, communication load, power consumption) following the Paraver trace format. Moreover, metrics are not hardwired on

[10] It is important to remark that data dependencies existing among tasks may limit the parallel execution. For example, when only 1 block is considered, the 31 created tasks are executed sequentially due to the data dependencies existing among them.
[11] http://www.bsc.es/computer-sciences/performance-tools/paraver

the tool but programmed by means of functions, allowing to display a large number of different metrics.

*Figure **6*** shows a snapshot of Paraver, representing the parallel execution time of the tasking model, considering that the frame is divided in 16 blocks and executed in the Xeon(R) CPU E5-2670 processor with 16-cores. Each bar in the figure represents the execution time of each function in one core (the same colour code presented *Figure **3*** is considered). We can infer the following conclusion: Despite the execution of each function is well balanced, dependencies towards *subtractPixelSides* function (purple bar) serialised completely the execution of the application, which reduces considerably the parallelism.

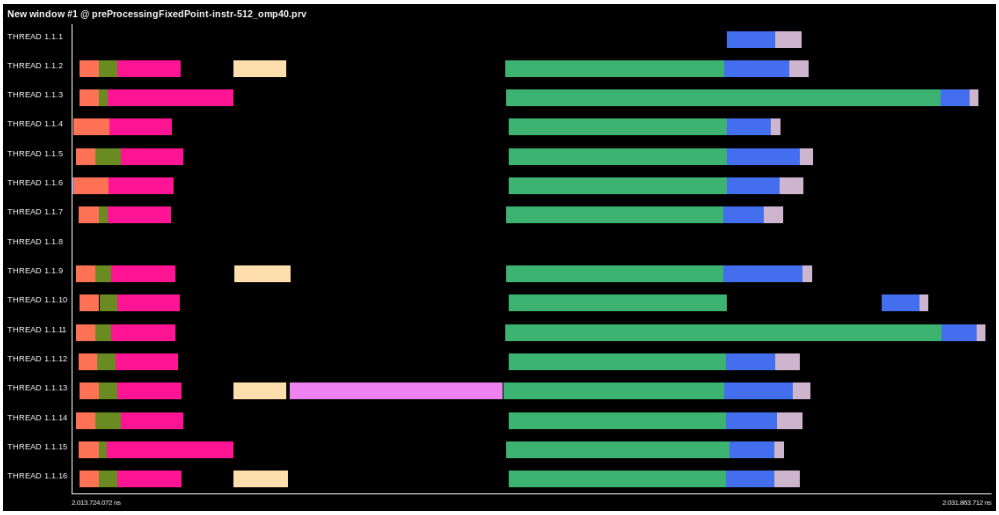

Figure 6. Paraver snapshot of the parallel execution of the space application.

Next section describes a new parallel version of the application to overcome the limitations observed in *Figure **6***.

5 Parallelisation Strategy 2

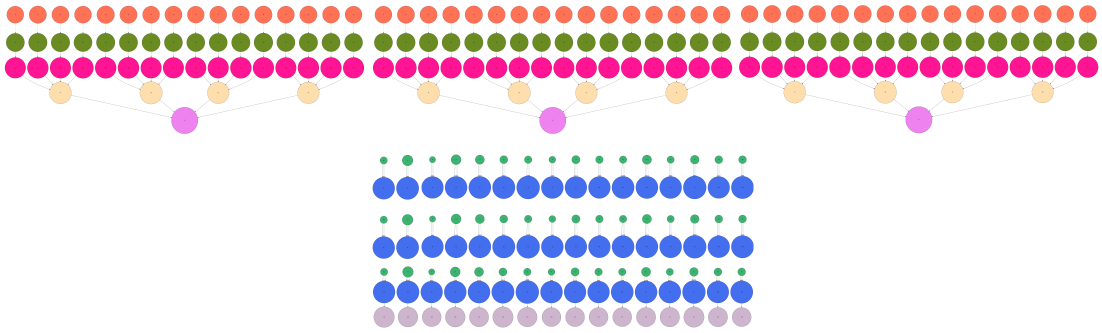

Figure 7. Application's TDG of the new parallelisation strategy in which the #Groups loop is parallelised.

In order to better use the core resources, we refine the previous parallelisation strategy by allowing execution in parallel of multiple iterations of the loop bounded by the *#Groups* parameter (see *Figure **1***). To do so, it is assumed that the system has *#Groups* sensor readouts ready to be processed by the application. *Figure **7*** shows the new TDG of the application, assuming that *#Groups* equals to 3.

It is important to remark that in order to efficiently reduce the overall application's response time, a working backlog needs to efficiently fill the cores. The simultaneous execution of multiple *#Group* iterations makes one single iteration to be delayed due to interferences with computation coming from other iterations. However, the overall application's response time is reduced as will be seen in the next section.

5.1 Performance evaluation

Figure **8** shows the performance speed-up of the new parallelisation strategy. As shown, the tasking model significantly increases the performance speed-up from 7x of 11x, compared to the previous parallelisation strategy. The reason is because the `depend` clause enables to efficiently synchronise the execution of multiple stages over different sensor readouts, while correctly fulfilling data dependencies.

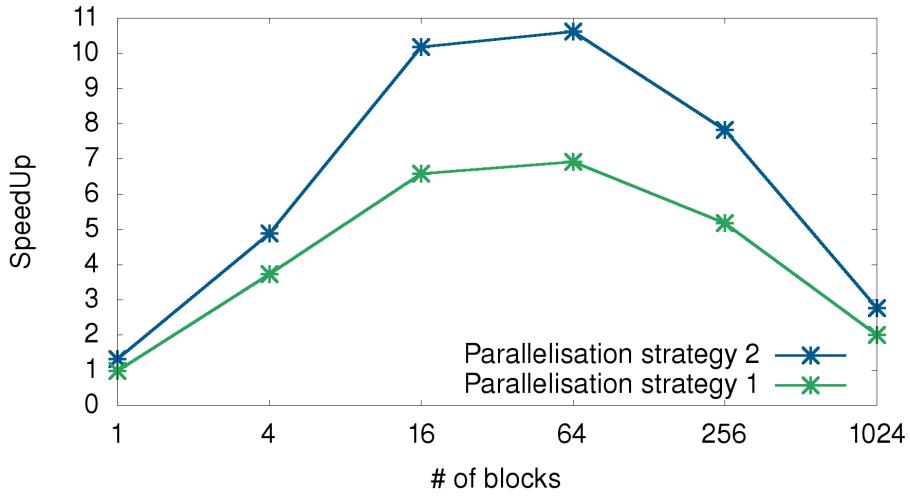

Figure 8. *Application's performance speed-up of the new parallelisation considering the tasking and fork-join execution models.*

5.2 Analysis of the parallel execution

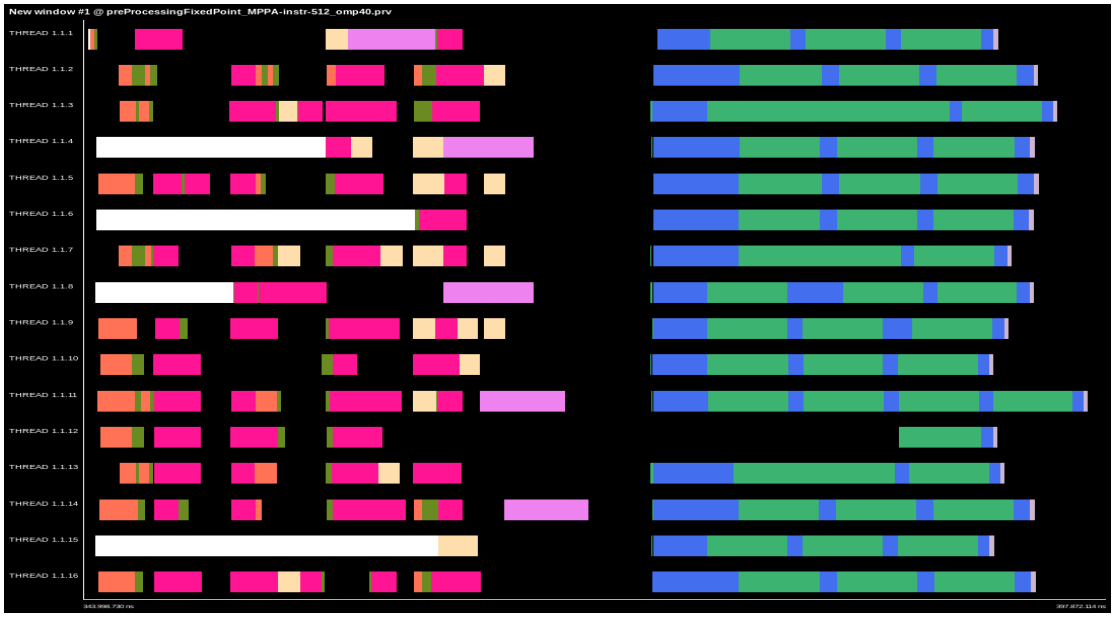

Figure 9. Paraver snapshot of the execution of new parallel version of the application.

*Figure **9*** shows a Paraver snapshot of the parallel execution of the new parallelisation strategy, using the same colour code as the one shown in *Figure **3***[12]. From this figure, we can observe that the use of the tasking model enables to better balance the parallel execution among cores, and so better performance can be achieved.

6 Time Predictability of OpenMP

The current specification of OpenMP lacks any notion of real-time scheduling semantics, such as deadline, period or *worst-case execution time* (WCET). Nevertheless, the structure and syntax of an OpenMP program have certain similarities with the *DAG-based scheduling model*[13,14]. This section evaluates the response time analysis of the parallel strategy 1 (see Section 3), considering an Intel(R) Core(TM) i7-4600U CPU running at 2.10GHz and featuring 4 cores.

6.1 OpenMP and DAG- based real-time scheduling

In scheduling theory, the *real-time task model*, either *sporadic* or *periodic*, is a well-known model to represent real-time systems. In this model, real-time applications are typically represented as a set of n recurrent tasks $\tau = \{\tau_1 , \tau_2 , .., \tau_n \}$, each characterized by three parameters: worst-case execution time (WCET), period (T) and relative deadline (D). In the *DAG-based scheduling model*, the representation of each task (called DAG-task) is enhanced by representing it with a directed acyclic graph (DAG) $G = (V,E)$, plus a period and a deadline. Each node $\upsilon \in V$ denotes a sequential operation or job, characterised by a WCET estimation. Edges represent dependencies between jobs: if $(\upsilon_1, \upsilon_2) \in E$ then the job υ_1 must complete its execution before job υ_2 can start executing. In other words, the DAG captures scheduling constraints imposed by dependencies among jobs, annotated with its WCET estimation.

Clearly the OpenMP tasking model resembles the DAG-based scheduling model: An OpenMP application corresponds to the DAG-task; nodes in G, i.e. jobs upon which timing analysis is derived, corresponds to OpenMP tasks; and edges in the G corresponds to dependencies defined with the `depend` clause. A method to construct the corresponding OpenMP-DAG from an OpenMP application is presented [15].

6.2 Response time analysis of the space application

In order to evaluate the time predictability of the space application, we compute the *response time upper bound* (R^{ub}) (also referred as *worst case makespan*[16]), which determines the maximum execution time of an application when considering the potential interferences the application may suffer when allocating all nodes from the DAG in a limited number of cores (taking into account dependencies).

[12] It is important to remark that the time scale in both Paraver snapshots is different, so shorter bar lengths does not mean shorter execution time of functions.

[13] Roberto Vargas, Eduardo Quinones and Andrea Marongiu, *OpenMP and Timing Predictability: A Possible Union?*, In DATE 2015

[14] Maria A. Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna and Eduardo Quiñones, *Timing Characterization of OpenMP4 Tasking Model*, in CASES 2015

[15] Roberto E. Vargas, Sara Royuela, Maria A. Serrano, Xavier Martorell, Eduardo Quiñones, *A Lightweight OpenMP4 Run-time for Embedded Systems*, in AspDAC 2016

[16] The *makespan* of a set of precedence constrained jobs, in our case OpenMP tasks, is defined as the total length of the schedule, i.e., *response-time*, of the collection of jobs

6.2.1 Work-conserving dynamic scheduler

When considering a work-conserving dynamic scheduler[17], as the one implemented in the OpenMP run-time from GNU-GCC (libgomp, see deliverable *D2.1 - Report on the evaluation of current implementations of OpenMP4* for further details), the response time upper bound can be computed by means of a schedulability test. In this project we consider the one derived in [14] and expressed as follows:

$$R^{ub} = \text{len}(G) + \frac{1}{m}\left(\text{vol}(G) - \text{len}(G)\right)$$

In which:

- *G* is the OpenMP-DAG extracted from the OpenMP application;
- *len (G)* is *critical path* of G, i.e. the longest chain in G;
- *vol(G)* is the sum of all WCETs of jobs in G.

The reasoning behind the schedulability test is the following:

- The *len(G)* factor determines the fastest execution time of the application, assuming an infinite number of available cores. Under this ideal scenario, nodes not belonging to the critical path can never interfere its execution, as there is always an available free core.
- The *(vol(G) - len(G))/m* factor corresponds to the even distribution of the workload not belonging to the critical path among the *m* cores available in the targeted architecture. The evenly distribution is only valid if a work-conserving scheduling strategy is considered.
- Overall, a response time upper bound adds to the *len(G)* factor, the maximum potential interference coming from the *(vol(G) - len(G))/m* factor.

6.2.2 Static scheduler

When the scheduler is statically defined, the core in which each node executes is predefined at system design. The use of static approaches is required in critical real-time embedded systems, such as avionics or automotive.

The response time minimization problem of DAG-based applications on parallel architectures is known to be NP-hard, and so sub-optimal static allocation approaches have been proposed [18]. The following priority rules have been shown to perform well in the context of parallel scheduling:

- *Longest Processing Time (LPT)*: the job with the longest WCET is selected;
- *Shortest Processing Time (SPT)*: the job with the shortest WCET is selected;
- *Largest Number of Successors in the Next Level (LNSNL)*: the job with the largest number of immediate successors is selected;
- *Largest Number of Successors (LNS)*: the job with the largest number of successors overall is selected;
- *Largest Remaining Workload (LRW)*: the job with the largest workload to be executed by its successors is selected.

6.2.3 Timing analysis of nodes in DAG

In order to derive the WCET estimation of each node in the OpenMP DAG (required to compute the response time), we measure the execution time of each OpenMP task in

[17] In a work-conserving scheduler, ready jobs are always executed if free cores are available.
[18] K. E. Raheb, C. T. Kiranoudis, P. P. Repoussis, and C. D. Tarantilis. *Production scheduling with complex precedence constraints in parallel machines*. Computing and Informatics, 24(3), 2012.

isolation, i.e. without suffering any software and hardware interferences, and increase it by a safety margin of 40%. This safety margin has been computed by observing the execution time variation of each node under execution scenarios with different level of interferences[19]. It is important to remark that in order to derive a trustworthy response time analysis, it would be required to compute the WCET estimation with a more accurate and sound timing analysis method. The objective of this section however, is to demonstrate the applicability of response time analysis methods to OpenMP applications, rather than investigating trustworthy WCET analysis methods for parallel computation.

6.2.4 Response time upper-bound

*Table **2*** shows the response time (in milliseconds) of the dynamic and static scheduling approaches presented above. Moreover, the Table also shows the actual execution time of the application, executed on the multi-core platform considered in this Section.

Actual Execution time	Work-conserving dynamic scheduler	Sub-optimal static schedulers				
		LPT	SPT	LNSNL	LNS	LRW
96	130.71	117.33	119.86	117.36	117.36	11801

Table 2. Response time analysis of the space application (in miliseconds).

Actual Execution time	Work-conserving dynamic scheduler	Sub-optimal static schedulers				
		LPT	SPT	LNSNL	LNS	LRW
96	130.71	117.33	119.86	117.36	117.36	11801

When considering a work-conserving dynamic scheduler, the response time upper-bound presented above is only 29% higher that the observed execution time. In case of static schedulers, the response time upper-bound is even tighter, being only 19% higher than the observed execution time. Overall, we conclude that OpenMP tasking model can be effectively timing analyse.

It is important to remark that the response time analysis does not take into consideration the run-time overhead. In fact, the purpose of this analysis is to demonstrate that OpenMP applications can be effectively analysed. Therefore, a more accurate and sound response time analysis is required to be investigated.

7 Exploring heterogeneous parallel execution

The latest specification of OpenMP (4.5, released on November 2015) introduces a new and powerful *acceleration execution model* (by means of the `target` directive), to support heterogeneous parallel computing, enabling OpenMP to exploit the huge performance opportunities of many-core heterogeneous platforms implementing distributed memory.

In order to explore the performance opportunities of a many-core heterogeneous architecture, in our case a Kalray MPPA processor architecture, we have developed a new parallel version of the application incorporating the `target` directive. The strategy adopted

[19] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega and Francisco J Cazorla, *Resource Usage Templates and Signatures for COTS Multicore Processors*, in DAC 2015

is to transfer to the accelerator devices of the selected architecture, a set block of frames to be processed in parallel. The purpose of this exercise is not to derive the most efficient parallelisation of the application, but to demonstrate the parallelisation opportunities brought by the OpenMP acceleration model.

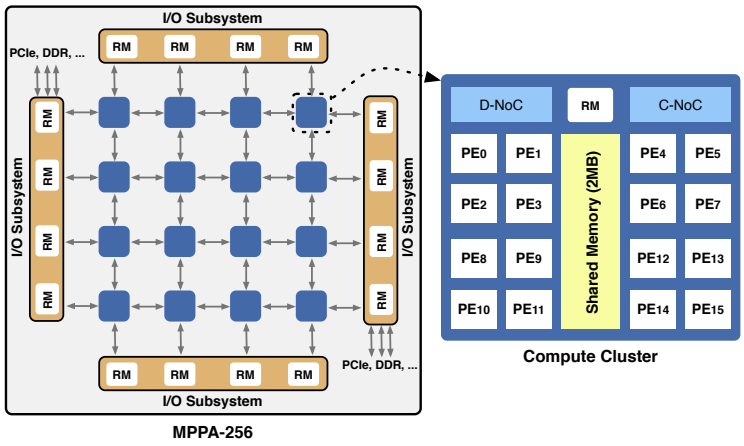

Figure 10. MPPA-256 processor architecture composed of 4 I/O subsystem (host) and 16 clusters (device).

Figure ***10*** shows a blocking diagram of the MPPA many-core architecture. The MPPA architecture incorporates *4 host subsystems* (I/O in MPPA nomenclature) composed of a 4-core processor each (cores in the I/O are named resource managers or RM), and a many-core fabric composed of *16 acceleration devices* (clusters in MPPA nomenclature) featuring 16-cores each connected to a 2MB on-chip memory (cores in the clusters are named processing elements or PM). This imposes a limitation on the data transfer and processes within a cluster, as only the sensor frame processed by the application occupies 8 MB.

Table ***3*** shows the memory requirements of input (*IN*), and input/output (*INOUT*) data dependencies required to compute each application stage, 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 reminds 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 2 and 4 blocks are simultaneously available in memory. This column uses the green and red colors to indicate when memory data structures that fits and do not fit respectively within the cluster local memory.

As shown, the memory requirements to process in parallel functions *detect-Saturation*, *subtractSuperBias*, *nonLinearityCorrection* among 4 blocks is around 1 MB, fitting within a cluster memory. In case of functions *detectCosmicRay*, *linearLeastSquares-Fit* and *calculateFinalSignal-Frame*, 4 blocks cannot be processed in parallel as it requires 3 MB, and so only 2 can be processed, requiring 1.5 MB of memory.

The *subtractPixelTopBotton* function fits within a cluster, requiring 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 case of *subtractReferencePixelSides* function, it cannot be executed within a cluster as it requires the full frame, and so it executes on the IO.

Table 3. 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					
		64-block Frame			Blocks per stages	
	Full Frame	1 block	2 blocks	4 blocks	2 blocks	4 blocks
2. detectSaturation		136 KB	272 KB	544 KB	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					1 MB (1 column composed of 8 blocks)	
INOUT currentFrame[N][N][BS][BS]	8 MB		1 MB			
6. subtractReferencePixelSides					8 MB (complete matrix needed)	
INOUT currentFrame[N][N][BS][BS]	8 MB		8 MB			
7. detectCosmicRay		832 KB	1,62 MB	3,25 MB	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	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		

*Figure **11*** shows the distribution of functions among the different clusters (devices) and IO (host) in a simplified version of the TDG processing 16 blocks, rather than 64.

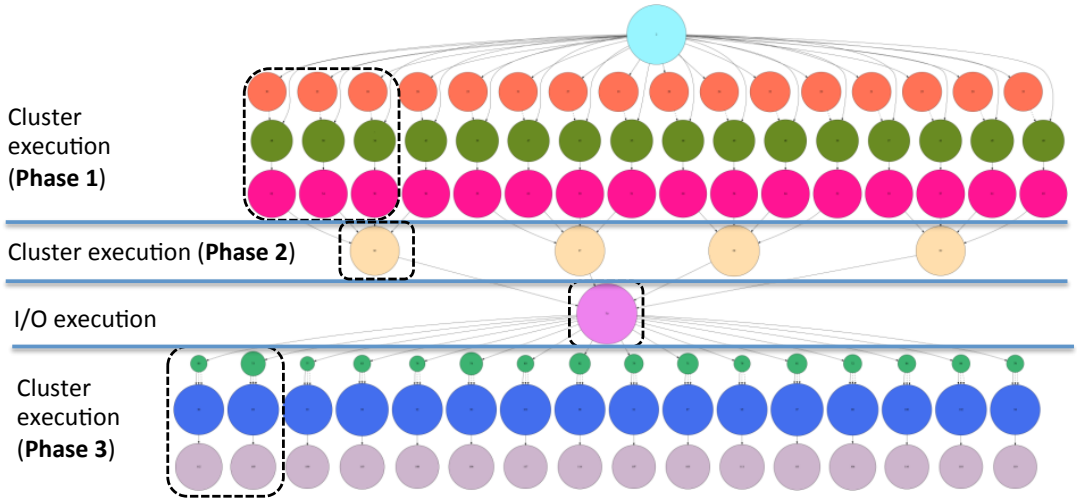

Figure 11. Parallel distribution of the application within the MPPA architecture.

7.1 Evaluation on the MPPA

Table **4** shows the execution time (in seconds) when running each of the execution phases described in *Figure **11*** (*cluster execution phases 1, 2* and *3*, and the *IO execution phase*) into a single IO core (labelled as *Sequential Time*), and when spawning each execution phase among multiple clusters (labelled as *Parallel Time*).

As shown the execution time drastically decreases, when the application is executed in parallel within clusters. Overall, we achieve a performance speed-up of 8.3x. Clearly, the IO execution phase, concretely the *subtractPixelTopBotton* function, seriously impacts the parallel execution time as already identified in *Figure **6***. In fact, this function represents the 50% of the overall execution. The table also presents the performance speed-up of each cluster execution phase, with speed-ups ranging from 11.41x to 22.17x.

Table 4. *Performance speed-up of the pre-processing sampling application when executing on a MPPA 256 processor architecture, considering the parallelisation strategy devised in Section 3.*

	Sequential Time (seconds)	**Parallel Time (seconds)**	**Speed-up**
CLUSTER-Phase 1	69.180	3.120	**22.17x**
CLUSTER-Phase 2	16.641	1.458	**11.41x**
IO-Phase	17.360	17.360	**1x**
CLUSTER-Phase 3	175.799	11.641	**15.10x**
Overall computation time	**278.980**	**33.579**	**8.3x**

8 Conclusions

In this document, we have evaluated the OpenMP tasking and acceleration execution model from a programmability, performance and time predictability point of view.

Regarding programmability, we have presented three parallelisation strategies for a pre-processing sampling application for infra-red H2RG detectors targeting both, homogeneous and heterogeneous parallel architectures, and demonstrating the capabilities of the OpenMP tasking and acceleration execution models.

Regarding performance, the parallelisation strategies using the OpenMP tasking model (named as strategies 1 and 2) have been evaluated from an average performance point of view, providing a speed-up of 7x and 11x respectively, on a 16-core homogeneous general purpose processor architecture (two Intel(R) Xeon(R) CPU E5-2670 processor, featuring 8 cores each). The parallelisation strategy using the accelerator model have been evaluated only from a average performance point of view on the 256 MPPA many-core heterogeneous architecture256 MPPA form Kalray, providing a speed-up of 8.3x.

Finally, *regarding time-predictability,* the OpenMP tasking parallelisation strategy 1 has been evaluated from a worst-case response time point of view on a 4-core homogeneous general purpose architecture (an Intel(R) Core(TM) i7-4600U), considering dynamic and static allocation approaches, providing a response-time upper-bound which is 29% and 19% respectively higher with respect to the average execution time.

Overall, we conclude that OpenMP is a very convenient parallel programming model for real-time embedded systems in general, and space systems in particular.

Annex II - D2.1. Report on the evaluation of current implementations of OpenMP

Paolo Gai, Eduardo Quinones
{pj@evidence.eu.com, eduardo.quinones@bsc.es}

June 2016

Dissemination level: All information contained in this document is public

Parallel Programming Models for Space Systems

(ESA Contract No. 4000114391/15/NL/Cbi/GM)

Abstract

This report evaluates *libgomp*, the current OpenMP4 run-time implementations from GNU GCC in terms of: (1) run-time overhead, (2) memory usage and (3) required operating system services.

Table of Contents

1 Introduction

Parallel programming models are a key element to efficiently exploit the parallel performance opportunities of multi-core architectures such as the NGMP GR740 considered in this project. Moreover, with the advent of many-core heterogeneous architectures (e.g. the 256 MPPA processor, also evaluated in this project), the use of parallel programming becomes mandatory to tame the complexity of programming such complex parallel architectures.

This project focuses on OpenMP, a parallel programming widely used in high-performance and general-purpose domain that addresses key issues in practical programmability parallel systems:

- It provides mature tasking execution model to support highly dynamic and irregular parallelism, augmented with features to express data dependencies among.
- It provides an acceleration execution model to efficiently couple the parallel execution occurring in the host and devices.

Deliverable *D1.1. Report on parallelisation experiences for the space application* evaluates the use of OpenMP is real-time systems considering three key metrics: programmability, performance speed-up and time predictability, and demonstrates that OpenMP is a very convenient parallel programming model to be used in real-time embedded domain in general, and in the space domain specifically.

This deliverable focus on evaluating the implementation of the OpenMP run-time. That is, current implementations of OpenMP have not been designed taking into account the potential execution environment constrains existing in embedded domains due to limited by hardware resources, operating systems (OS) or application requirements. This is the case, for example, of on-board space systems targeting the NGMP GR740 processor, the RTEMS-SMP OS and applications with a limited working set.

Instead, current implementations, e.g. libgomp from GCC or nanos++ from OmpSs, target general-purpose processors based on linux-like OS. The corresponding run-time implements large and complex data structures, which result in an important run-time overhead that must be compensated with large working set applications. There is therefore a necessity to implement efficient lightweights OpenMP run-times in which the overhead is reduced to the bare minimum.

The rest of the document is organized as follows. Section 2 describes the current implementations of the OpenMP software stack. Section 3 evaluates the efficiency, in terms of performance and memory usage, of current OpenMP run-time implementations. Section 4 describes the OSs used in the embedded domain, and proposed a set of new OS services required to execute OpenMP. Finally, conclusions are presented in Section 8.

2 The OpenMP software stack

This project considers the OpenMP implementation provided by GNU GCC, known as GOMP[20]. GOMP is composed of four main components[21] (see *Figure **12***):

[20] Gcc GOMP, https://gcc.gnu.org/projects/gomp/

[21] Diego Novillo, "OpenMP and automatic parallelization in GCC", Proceedings of the GCC Developers Summit, 2006

1. *Compiler*, in charge of validating correct syntax and semantics of OpenMP directives and clauses as defined by the OpenMP specification, and emitting the corresponding call to the OpenMP run-time API.
2. *API run-time*, in charge of providing the run-time services required to implement OpenMP directive and clauses.
3. A *high-level library* (named *libgomp)*, containing the API implementation of all user-level functions specified by the OpenMP standard, as well as the GCC-specific functions to implement the semantics of the various directives. This library uses an internal abstract notion of low-level constructs for threading and synchronization.
4. A *low-level library*, providing the actual implementation of the higher-level constructs on the target platform, usually relying on available services provided by the operating system (OS).

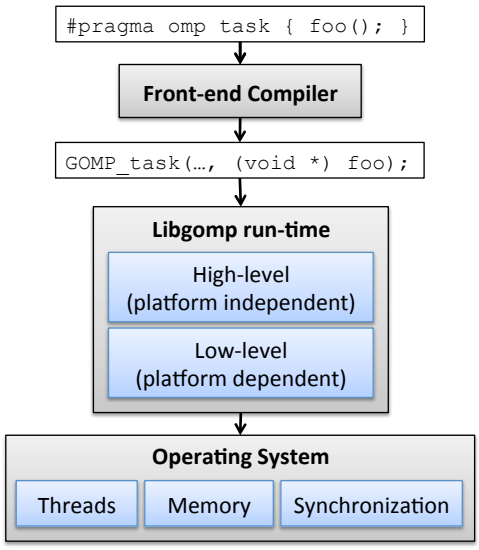

Figure 12. OpenMP software Stack.

Next sections discusses the libgomp run-time, evaluating the performance overhead and memory usage, and discusses on the OS services required to execute it.

3 Evaluation of the OpenMP Run-time

The OpenMP tasking model provides a very convenient abstraction for describing fine-grain and irregular parallelism, in which the run-time is in charge of scheduling tasks to threads[22]. Unfortunately, the management of tasks introduces an extra overhead in the run-time library, which may diminish the benefits brought by parallel execution, when the number of tasks managed at run-time is too high. Next section evaluated the run-time overhead and memory usage of the tasking model.

3.1 Tasking model overhead

*Figure **13*** shows the performance speed-up of the pre-processing sampling application for infra-red H2RG detectors considered in this project, applying the parallelisation strategy 2 in which *#Groups* loop iterations are executed in parallel (see Section 4 of deliverable *D1.1. Report on parallelisation experiences of the space application* for further details), and executing with two OpenMP run-time libraries: *libgomp* from GCC GNU and *Nanos++* from OmpSs[23]. In order to better understand this figure it is important to consider *Table **5*** (already presented in deliverable D1.1), which relates the number of blocks in which the sensor frame is divided, with the number of tasks created, and the size of each block processed by tasks.

[22] See deliverable *D1.1 - Report on parallelisation experiences of the space application,* for further information

[23] OmpSs is a parallel programming model compatible with OpenMP and developed by BSC whose objective is to investigate on *asynchronous* parallelism and *heterogeneity* (devices like GPUs). The framework is composed of the source-to-source compiler *Mercurium* compiler and the *nanos++* runtime system.

Table 5. Number of created tasks at run-time and size of the sub-frame processed by each task, when dividing the sensor frame in blocks.

Number of blocks	Created tasks	Block size
1	31	2048x2048
4	114	1024x1024
16	436	512x512
64	1704	256x256
256	6736	128x128
1024	26784	64x64

As shown, nanos++ obtains the peak performance when each task operates over an input working set of 512x512 elements (436 tasks are created, see Table 1). Instead, libgomp enables tasks to operate over a smaller working set (256x256 elements), while still increasing performance.

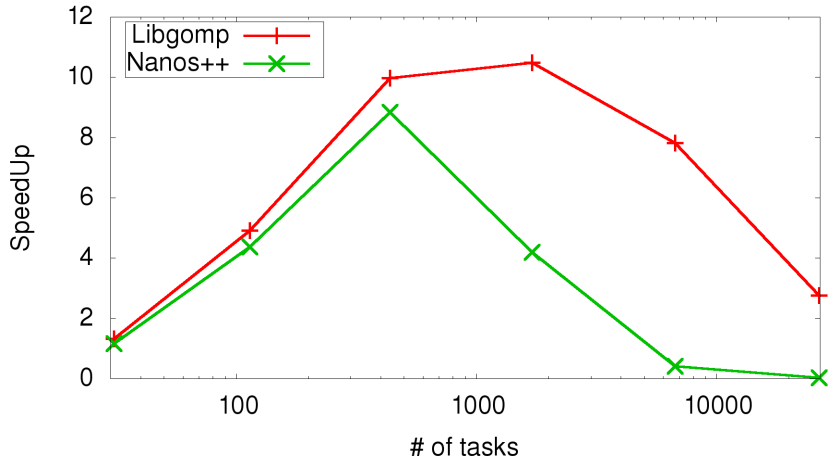

Figure 13. Performance speed-up of the space application when executing with two OpenMP4 runtimes: libgomp and nanos++.

In order to ensure that tasks are executed in the correct order as imposed by the `depend` clause, libgomp and nanos++ build the TDG at run-time for a twofold reason: (1) the TDG depends on the tasks that are instantiated (and so executed), which is determined in turn by the control flow graph (CFG); and (2) the addresses of the data elements upon which dependencies are build, are known at run-time.

Hence, when a new task is created, its `in` and `out` dependencies are matched against those of the existing tasks. To do so, each task region maintains a *hash table* that stores the memory address of each data element contained within the `out` and `inout` clauses, and the list of tasks associated to it. The hash table is further augmented with links to those tasks depending on it, i.e. including the same data element within the `in` and `inout` clauses. In this way, when a task completes, the run-time can quickly identify its successors, which may be ready to execute.

As shown in Figure 1, libgomp makes a more efficient use of the hash-table than nanos++ as the number of tasks increases. The reason is because nanos++ allows defining dependencies between two arrays in which their elements are partially overlapped, introducing an extra overhead when the number of tasks is very high. This property further facilitates programmability, as it is not require identifying which exact elements overlap, but simply

defining a dependency between arrays. However, the programmability benefits are diminished by the performance degradation when small working sets are considered.

Overall, in order to exploit fine-grain parallelism in which tasks can operate over small working sets, it is fundamental to develop lightweight run-times with very small performance overhead.

3.2 Memory Consumption

Building the TDG at run-time requires storing the hash tables in memory until a `taskwait` or a `barrier` directive is encountered. Since dependencies can only be defined between sibling tasks, when such directives are encountered, all tasks in their binding region are guaranteed to finish. Removing the information of a single task at completion would result too costly, because dependent tasks are tracked in multiple linked lists in the hash table. As a result, the memory consumption may significantly increase as the number of instantiated tasks between two synchronization directives increase as well.

Such memory consumption is not a problem in general-purpose systems, in which large amounts of memory are available. However, this is not the case in the newest many-core embedded architectures. For example, the MPPA processor integrates 16 clusters of 16-cores each, with a 2 MB on-chip private memory per cluster. Despite the overall size of the MPPA memory is 32 MB, clusters only have access to their private memory. The rest of memory is accessible through DMA operations (with a significant performance penalization), and so it is highly recommended that the complete program (including the OpenMP run-time library) resides within the private memory.

*Figure **14*** shows the memory consumed by libgomp at run-time when applying the parallelisation strategy 1, and new parallel version in which the `taskwait` directives have been replaced by *fake* dependencies that emulate the exact same functionality, with the objective of artificially increasing the memory requirements of the hash-table, by maintaining it until the end of the program (labelled as *dummyDeps*). The memory usage has been extracted using Valgrind Massif [14] tool, which allows profiling the heap memory consumed by the run-time in which the TDG structure is maintained.

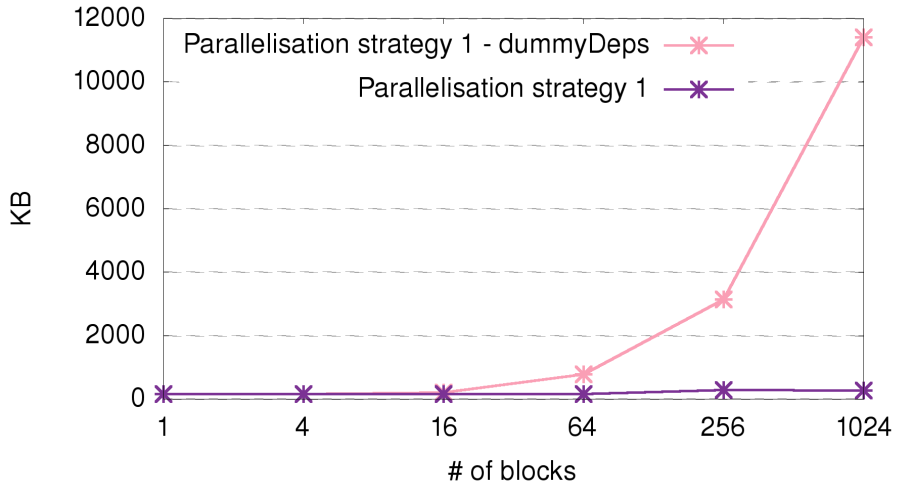

Figure 14. Heap memory consumption due to maintain the TDG data structure at run-time.

As the number of instantiated tasks increases, the memory consumed by the run-time increases as well. However such increment is very small in case of the *Parallelisation strategy 1,* since the `taskwait` directive releases the hash-table structures from time to time. This is not the case when the `taskwait` directive is replaced by fake dependencies,

as it is shown in the *dummyDeps* curve, in which the memory used rapidly increases as the number of instantiated tasks increases as well.

*Figure **15*** shows the performance speed-up obtained for each of the parallel versions analysed in *Figure **14***. As expected the extra dependencies introduced by *dummyDeps* harms the performance speed-up.

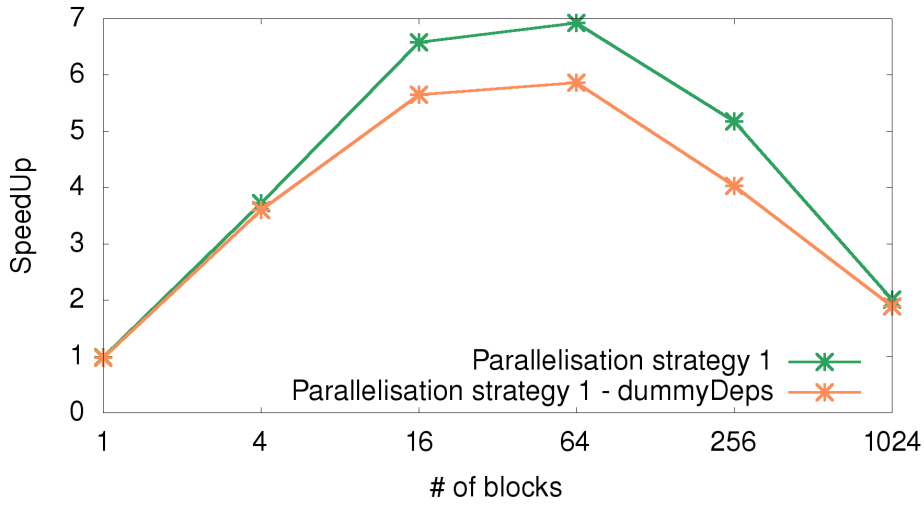

Figure 15. Performance speed-up of the three parallelisation strategies when the number of instantiated tasks increases.

4 Services to support OpenMP low-level run-time

The libgomp run-time is fully supported on Linux, which essentially implements a wrapper around the POSIX threads library[24] (with some target-specific optimizations for systems supporting lighter weight implementations such as the use of *locking* primitives implemented with atomic instructions and *futex* system calls), and provides support for thread-local storage. In fact, with few exceptions, most of the synchronization mechanisms implemented in libgomp are "simply" a direct mapping to the underlying POSIX routines.

Therefore, any OS supporting POSIX API already provides (in principle) the set of functionalities required to support the OpenMP software stack. One example is RTEMS[25] that indeed, already supports the OpenMP programming model [26]. Deliverable *D3.1. Report on the applicability of OpenMP4 on multi-core and many-core space platforms,* discusses on the support that RTEMS provides on OpenMP.

This section evaluates the services that a non-compliant POSIX OS requires to support OpenMP. OS not supporting POSIX are commonly minimal OSs used in constrained critical real-time systems such as avionics and automotive domain, implementing tiny and specialized API. Among them, we can find ERIKA Enterprise[27], a minimal OS developed by partner Evidence that provides hard real-time guarantees. This report considers ERIKA in the explanation to better illustrate our ideas.

[24] POSIX Threads, https://en.wikipedia.org/wiki/POSIX_Threads
[25] RTEMS RTOS, https://www.rtems.org
[26] OpenMP RTEMS project, https://devel.rtems.org/wiki/OpenMP
[27] Erika Enterprise RTOS, http://erika.tuxfamily.org

4.1 Minimal OS vs. POSIX pthreads

The structure of the free and commercial OS existing in the market is heavily influenced by the OS API they implement, especially those OS targeting real-time (RTOS). If we want to make a classification, we can roughly divide them in three areas described below.

4.1.1 POSIX compliant RTOS

In this area, we include all the RTOS and general-purpose systems implementing the POSIX specification, either in its complete form or in a reduced form (such as the IEEE 1003.13 PSE51 specification).

The main characteristic of this class of kernels is the availability for the user of a number of API, that include an “almost” complete libC (PSE51 kernels may lack a subset of the specification), and at least the pthread API.

As a reference number, a system implementing a complete PSE51 kernel will include the implementation of a few hundred functions, with a code footprint of at least 50 KB. Among those functions, we want to cite the dynamic creation of threads, mutex, semaphores, etc., as well as the dynamic creation of signal handling, memory allocation, and device drivers accessed often by means of a device file-system.

Linux and RTEMS are part of this group of kernels.

4.1.2 Dynamic RTOS

In this context, with *Dynamic RTOS* we intend a minimal RTOS that tries to implement a rich set of primitives in a custom way, often mimicking their POSIX “big cousins".

Among the primitives offered by those kernels, we cite thread/semaphores/message queues dynamic creation, (sometimes) a tiny memory allocator, and a set of device drivers to directly access the microcontroller peripherals. The scheduler is typically a priority scheduler, sometimes with limited support for round robin threads (as opposed by the more complex POSIX real-time scheduler available in the bigger kernels).

These kernels often are available for small microcontrollers, and their target is to provide flexibility of programming (thanks to the rich API) with a reasonable footprint (in the order of 5-20 Kb).

Examples of this group of RTOS are FreeRTOS, embOS, MQX, ChibiOS, and others.

4.1.3 Static RTOS

With a *Static RTOS* we intend a further reduction in flexibility in the operating system API, meant to target low cost and hard real-time scenarios. The typical example of this group is the automotive standard API OSEK/VDX as well as its extension made in AUTOSAR OS.

This class of kernels are tiny schedulers, which implements an API composed by a few tens of functions. They basically implement a minimal fixed priority scheduler with immediate priority ceiling support, plus periodic activation and limited support for blocking primitives. Memory allocation is not available, and dynamic creation of objects is forbidden (all the configuration is handled using special configuration languages, parsed by specific configuration tools).

Another notable feature available is that the system can implement a single stack configuration, with threads sharing their stack thanks to a run-to-completion semantic. This allows a further reduction on the memory footprint, which is typically in the range of 2 to 5 KB.

An example of this class of kernels is ERIKA Enterprise, analysed in this project, and which is shortly described in the next subsection.

4.2 Erika Enterprise

ERIKA Enterprise is the first open source RTOS which has been certified OSEK/VDX compliant. The kernel is available under the GPL2+Linking Exception, and is currently mainly maintained by Evidence and the ReTiS Lab at the website: http://erika.tuxfamily.org

ERIKA Enterprise strictly follows the OSEK/VDX automotive standard. This standard is meant for the automotive market and therefore aims at reducing memory consumption (i.e., footprint, typically of a few KBs), run-time latencies and error-prone conditions. For these reasons, most configuration settings (including the total number of threads and their priorities) are statically defined at compile-time through a specific configuration file (named OIL file). Moreover, unlike Linux, the set of services provided by the RTOS is very small and simple compared to the one defined by the POSIX standard.

In addition to the OSEK/VDX standard, ERIKA Enterprise supports additional research schedulers such as EDF, IRIS, and hierarchical schedulers.

ERIKA Enterprise is currently being used in production in various automotive applications (for gasoline injection, gearboxes, parking sensors, electric motors) by companies such as Magneti Marelli, Vodafone Automotive, Piaggio, and others. Moreover, ERIKA Enterprise is currently used by various research projects (most of them unrelated to Evidence) including: P-SOCRATES (FP7), PROXIMA (FP7), HERCULES (H2020), ARAMIS, AMALTHEA (ITEA), INCOBAT, EDAS, and others.

The internal implementation of ERIKA Enterprise divides the kernel code in two main parts, one named “CPU”, which is related to the implementation of a hardware abstraction layer, including context change, interrupt handling routines, and other architecture dependent code, and a “kernel” layer, which implements the scheduling policies following the OSEK/VDX API. The kernel currently supports more than 10 CPU architectures (from small 8 bitters to 32 bit multicores for automotive), including the following: AVR 8bit, MSP430, ARM Cortex M0/3/4 and R4, Altera Nios II, Lattice Mico32, EnSilica esi-RISC, Freescale S12, Microchip dsPIC, PIC24, and PIC32, Renesas RX200, X86, Infineon Tricore AURIX 26x, 27x, 29x, PowerPC e200 z0, z4, z6, z7 (various single and multicore chips supported) and the many-core accelerator fabric (cluster part) of the 256 MPPA. ERIKA supports more than 15 compilers, including GCC, Diab, Hightec, IAR, Cosmic, Tasking variants on specific microcontrollers.

ERIKA Enterprise supports multicores by means of a static partitioning of threads into cores (this is similar to what done by the AUTOSAR OS specifications). In other words, threads are statically assigned to cores at compile time. The OS objects are visible in all cores, in a way that a primitive acting on an object allocated on another core (e.g., a task activation of a task allocated on another core) results in a multicore interrupt plus a minimal communication performed using spin locks.

As for the spin locks, they are implemented (depending on the version of the OS) either without policy or using a FIFO policy with the G-T algorithm[28]. As the kernel implements a partitioned strategy, there is limited usage of the spin locks in the kernel code, which is basically limited to the inter-processor interrupt handling and to an initial barrier at kernel start-up.

[28] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. IEEE Computer, 23(6):60-69, June 1990.

4.3 ERIKA Enterprise and libgomp

In order to support libgomp on top of an OS such ERIKA Enterprise, it is required to work on two components: the lower-level library component and the OS (see *Figure **12***). The former maps the high-level OpenMP constructs to the OS services; the latter ensures that the OS exports the minimal set of services needed by a libgomp run-time environment.

To do so, there exist two main approaches to practically implement the OpenMP semantics in a non-POSIX OS like ERIKA Enterprise:

i. *Dynamic creation of threads at run-time.* This approach is flexible but expensive in terms of both memory usage and execution time[29]. In a resource-constrained environment like embedded, this approach may easily run out of memory with run-time overheads diminishing fine-grained parallelism. Moreover, this approach is not in line with the static structure of a minimal OS like ERIKA Enterprise.

ii. *Static creation of threads.* This approach enables the creation of lightweight threads (typically as many as the number of processors) at system start-up, and "docks" them on a "pool" of idle workers.

4.4 A proposal of the services required

Assuming an implementation based on thread pools (with static creation), our preliminary investigation has identified the following missing services that an OS such an ERIKA Enterprise must implement to support the OpenMP semantics:

- Thread management primitives for *creating, activating* and *joining* a job.
- Synchronization for a data structure representing a *spinlock* and related primitives for *locking*/*unlocking*; and data structures representing a condition variable and related primitives for *waiting*/*notifying*.
- For what concerns memory management primitives, we can rely on typical primitives (i.e., *malloc* and *free*) provided by the standard libc.

In the specific case of ERIKA Enterprise, these primitives required a reimplementation of the kernel layer, to support the OpenMP needs. The memory management primitives are not used internally by the kernel primitives, and they have limited usage in the OpenMP layer. We did not do a complete evaluation of the usage of the memory primitives during this project, however an initial implementation of them has been performed as part of the FP7 P-SOCRATES Project[30]. A complete evaluation of the memory usage primitives is left as part of future work.

Next, we provide a possible signature of the system calls to be incorporated in the OS. Even if we take into account ERIKA Enterprise, the approach is generic enough to be adopted with small changes for the other RTOSs of the same size.

```
StatusType CreateJob ( JobType*      JobId,
                       int           TaskId,
                       int           JobPrio,
                       void          (*JobRoutine)(void*),
                       void*         JobArg,
                       int           StackSize);
```

[29] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos, "A microbenchmark study of OpenMP overheads under nested parallelism", IWOMP 2008
[30] www.p-socrates.eu

Where:

- *JobId* is an output argument containing the unique identifier used to refer to the *job*.
- *TaskId* is an input argument containing the identifier of the static task of the RTOS that will execute the function.
- *JobPrio* is an input argument containing the propriety of the job.
- *JobRoutine* is the function to be executed.
- *JobArg* is the argument to be passed to the function JobRoutine.
- *StackSize* represents the stack size allocated for the job.

When **CreateJob** is called, the task descriptor data structures are created by the OS. Initially, newly-created tasks are configured to be in a suspended state. The tasks can then be activated using **ActivateJob**:

```
StatusType ActivateJob (JobType JobId);
```

When a task is activated, it starts executing the function associated with the including job. **JoinJob** is used to wait for *job* completion: this function returns when all the tasks associated with the job are completed, i.e. have returned from JobRoutine.

```
StatusType JoinJob (JobType JobId);
```

The low-level synchronization primitives designed and implemented within ERIKA for integration with the OpenMP runtime library are described in the following.

```
void SpinLockObj (SpinLockObjType* SpinLockObjRef);
void SpinUnlockObj (SpinLockObjType* SpinLockObjRef);
```

These primitives have a direct implementation over the spin lock primitives available in the CPU layer of ERIKA (see Section 3.2).

Besides locks, low-level ERIKA primitives also provide WAIT/SIGNAL synchronization, which operate on a **BlockableValueTypeRef** data type:

```
void InitBlockableValue(BlockableValueTypeRef BlockableValueRef,
                        TypeValue             Value);
```

This function initializes the *BlockableValueRef* instance for use with WAIT/SIGNAL API, and sets its initial value to *Value* (unsigned 32bit).

```
StatusType WaitCondition(BlockableValueTypeRef BlockableValueRef,
                         WaitCondType          WaitCond,
                         TypeValue             RightValue,
                         BlockPolicyType       BlockPolicy);
```

WaitCondition implements a blocking operation on a given condition. Each field is described as follows:

- *WaitCond* represents the condition operator; the accepted values are:
  - **VALUE_EQ**
  - **VALUE_NOT_EQ**
  - **VALUE_LT**
  - **VALUE_GT**
  - **VALUE_LT_OR_EQ**
  - **VALUE_GT_OR_EQ**
- *RightValue* is the value used on the right hand side (RHS) of the condition check expression.
- The *BlockPolicy* parameter is used to specify the wait policy:

- **BLOCK_NO** (WAIT_BUSY) the condition is checked in a busy waiting loop;
- **BLOCK_IMMEDIATELY** (WAIT_SLEEP) the condition is checked once. If the check fails (and no other tasks are available for execution) the processor enters sleep mode until the condition is reached. A specific signal is then used to wake-up the processor.
- **BLOCK_OS** (WAIT_TO_OS) informs the OS that the ERIKA task (i.e., the OpenMP thread mapped to that task) is voluntarily yielding the processor. The OS can then use this information to implement different scheduling policies. For example, the task can be suspended and a different task (belonging to a different *job*) can be scheduled for execution. When a task is suspended, a reference is pushed into a queue accessible as a field in the BlockableValue structure. The blocked task are unblocked when the BlockedValue is signalled.

Once the wait condition has been reached, the proper signalling response must be issued.

```
StatusType SignalValue(BlockableValueTypeRef BlockableValueRef,
                       TypeValue             Value);
```

This function posts a new *Value* into the **BlockableValue** referenced by *BlockableValueRef*. When this function is called, all the tasks waiting on the associated **BlockableValue** are notified. The notified tasks can verify again that their wait condition is satisfied. If this is not the case, they are blocked again (in the **WaitCondition** function body, where the condition is checked).

4.5 Estimation of hardware resources

In order to evaluate the impact that these new functions would have on the memory footprint, we consider again the ERIKA Enterprise as a baseline, due to its minimal footprint ranging between 1KB and 4 KB. Maintaining the memory footprint to the bare minimum is an important requirement in these type of OSs. Concretely, we have estimated the memory requirements needed to implement these additional services have considering a thread pool implementation, and a single ELF for multi-core support (which is implemented as part of the ERIKA3 implementation effort).

Table 2 presents the additional memory consumption in terms of extra memory used and code footprint of including the required OS services into ERIKA Enterprise to support OpenMP execution, with a reference to the Kalray MPPA architecture considered in this project.

Table 2. Extra memory used and code footprint of including the required OS services into ERIKA Enterprise to support OpenMP execution.

Description	Additional memory
Code footprint	1024 – 2048 bytes
RAM usage	128 bytes for each core

5 Conclusions

In this document, we have evaluated current OpenMP implementation in terms of run-time overhead and memory usage, and the OS services required to be executed in an embedded constrained environment.

Regarding the run-time overhead, we have evaluated two OpenMP implementations (nanos++ and libgomp). Our evaluations show that the performance benefits of parallel computation diminishes when tasks operates over blocks smaller than 512x512 in case of nanos++ and 256x256 in case of libgomp. Moreover, an intensive usage of dependencies may make the memory consumption to significantly increase.

Regarding the OS services required to execute the OpenMP run-time, we have first classified current RTOS in three different categories (POSIX compliant, dynamic RTOS and static RTOS), depending on the targeted embedded environment. For those RTOS targeting the highest critical systems (static RTOS), we have proposed a set of OS services which a minimum code footprint and memory requirements that enables the execution of OpenMP programs in critical real-time environments.

Overall, we conclude that: (1) the OpenMP run-time can be supported on static RTOS, and (2) the OpenMP run-time can diminish the performance benefits brought by parallel computation, and so investigations on low-overhead and lightweight OpenMP run-times is desirable.

Annex III - D3.1. Report on the applicability of OpenMP4 on multi-core space platforms

Michele Pes, Maria Serrano, Paolo Gai, Eduardo Quinones
{m.pes@evidence.eu.com, maria.serranogracia@bsc.es, pj@evidence.eu.com, eduardo.quinones@bsc.es}

June 2016

Dissemination level: All information contained in this document is public

Parallel Programming Models for Space Systems

(ESA Contract No. 4000114391/15/NL/Cbi/GM)

Abstract

This report describes the experiences of compiling RTEMS operating system and OpenMP for a quad-core LEON3 and a NGMP GR740 processor architectures. Moreover, this report evaluates the performance speed-up of the two OpenMP parallel versions of the pre-processing sampling application considered in this project (and devised in deliverable D1.1) on the two processor architectures.

Table of Contents

1 Introduction

Multi-core systems are considered as the solution to cope the performance requirements of current and future real-time (RT) embedded systems. In the space domain, the use of multi-core processor architecture is motivated by the development of the NGMP GR740 from Cobham-Gaisler, a quad-core 32-bit fault-tolerant LEON4FT SPARC V8 processor. In this same direction towards the adoption of multi-core, the latest versions of RTEMS operating system includes SMP (symmetric multi-processor) capabilities to support to execute on a shared memory homogeneous multi-core architecture such as the GR740. However, the appropriate parallel programming models to fully exploit the performance capabilities of the NGMP are not yet adopted in the space domain.

OpenMP is a well-known parallel programming model in the high-performance domain for shared memory architectures that implements a very powerful tasking model, to efficient exploit fine grain and irregular parallelism. Deliverable *D1.1 - Report on parallelisation experiences for the space application,* evaluates the OpenMP tasking models from a programmability, performance and time predictability point of view, considering a general-purpose processor architecture (Intel-based).

In this report we present our experiences of compiling the OpenMP tasking model on a *space platform* composed of the RTEMS SMP OS and two LEON-based multi-core architecture, i.e. a quad-core LEON3 and a NGMP GR740, considering the two OpenMP parallel versions of the pre-processing sampling application presented in deliverable D1.1.

The rest of the document is organized as follows: Section 2 presents the experiences of building RTEMS SMP with the GNU-GCC implementation of OpenMP. Section 3 presents the performance evaluation of the space application executed on two LEON-based multi-core architectures: a quad-core LEON3 and a NGMP GR740. Finally, conclusions are presented in Section 4.

2 Experiences on building RTEMS + OpenMP

2.1 RTEMS v4.11 + *GCC* 4.9.3

We initially started compiling RTEMS v4.11, as it was the most stable version at that time (January 2016). We cross-compiled RTEMS v4.11 with *GCC* 4.9.3 (supporting OpenMP v4.0), including the built-in support for SparcV8 LEON3, using the RTEMS Source Builder (RSB).

Due to the limited access to a multi-core LEON board, we performed the initial evaluations with the *Tsim* LEON simulator environment provided by Cobham-Gaisler, and compatible with SparcV8 ISA, to validate the functional correctness of the compilation. However, note that the *Tsim* simulator does not support parallel execution.

The initial problem we faced was the incompatibility of defining OpenMP directives into RTEMS tasks, including the RTEMS entry point task named *init*, resulting in the following run-time error "`libgomp: could not create thread pool destructor`". After a deep analysis of libgomp source code (the GNU GCC OpenMP run-time), we fixed the error by adding the following two macros into the application:

```
#define CONFIGURE_MAXIMUM_POSIX_KEY_VALUE_PAIRS 10
#define CONFIGURE_MAXIMUM_POSIX_KEYS 10
```

These macros enabled the run-time to allocate the resources required by the OpenMP execution environment. Despite the error was not shown anymore, libgomp was stalled in a deadlock situation, concretely in the run-time *sem_post* semaphore.

The same error was observed when using pthreads only (and so compiling with *pthread* rather than *fopenmp* flag). We tested an application spawning 2 threads and 1 RTEMS task. While the two threads ran regularly, the RTEMS task was not executed.

The same test cases (with both OpenMP directives and pthreads) executed correctly when no RTEMS tasks were created (this can be done by specifying the "*main*" or "*Posix_Init*" functions as application entry point, and so avoiding the creation of any RTEMS task).

The RTEMS documentation specifies that when compiling with the *-fopenmp* flag, the *rtems.h* file cannot be included in the application. In order to validate it, we performed a test in which some RTEMS API calls that required the inclusion of *rtems.h*, and compiled using *-pthread* and *-fopenmp* flags, and no problems were detected. We cannot exclude however, that more complex application using a different set of API could not be able to run under these conditions.

2.2 Experiences on RTEMS v4.12 + *GCC* 6.0.0

After contacting with RTEMS developers, they recommended us to abandon v4.11 and start evaluating v4.12, despite not being a stable version under constant development with the master branch frequently updated (developments on 4.11 branch seemed to be dormant/stopped).

The same steps done to compile RTEMS 4.11 were followed to compile RTEMS 4.12, but building on top of GCC6.1.1, which supports the latest version of OpenMP 4.5. As usual, we cross-compiled the tool-chain for sparcV8 LEON3 with the help of RSB. We performed the same experiments as in version 4.11, and the same error appeared, i.e. the incompatibility of OpenMP directives and RTEMS tasks.

We had access for a limited amount of time to a quad-core LEON3 board. A single core was detected by both RTEMS and OpenMP APIs, and so no parallel computation was performed.

2.3 Experiences on RTEMS v4.12 + *GCC* 6.1.1 + *enable-smp* flag

After discussions with RTEMS developers and colleagues from ESA, we discovered the existence of the undocumented flag *--enable-smp* when configuring RTEMS. Table 1 contains the commit version of the RTEMS Source Builder (RSB) and the RTEMS OAR master branch.

Table 6. RTEMS Source Builder (RSB) and the RTEMS OAR master branch information used.

	RTEMS Source Builder (RSB)	RTEMS OAR master branch
Commit	3da4d0e5ce50909d4fe45168d6d6a82ebc119f92	c6556e2ecc6b80f981bb210d541544f24b7f59df
Date	Mon May 30 15:07:41 2016 +0200	Wed Jun 1 14:38:05 2016 +0200
Author	Sebastian Huber <sebastian.huber@embedded-brains.de>	

When using this flag, applications cannot mix RTEMS APIs with OpenMP directives. In order to overcome this, we separated the RTEMS part from the OpenMP part in two separated source files. Moreover, we had to replace the standard `int main()` entry point, by the RTEMS Init task in order to make it run.

3 Evaluation on the pre-processing sampling application on space processor architectures: a quad-core Leon 3 and a NGMP GR740

*Table **7*** shows the performance speed-up of the two parallelisation strategies devised for the pre-processing sampling application (see deliverable *D1.1 - Report on parallelisation experiences of the space application* for further details) when executing on a quad-core LEON3 processor architecture implemented on a FPGA running at 80 Mhz, and a GR-CPCI-GR740 Quad-Core LEON4FT Development Board[31] running at 250 Mhz. The parallelisation considers that the frame is divided in 64 blocks.

Table 7. Performance speed-up of the two parallelisation strategies presented in D1.1, when executing on a quad-core LEON3 and a NGMP GR740 processor architectures.

Multi-core processor architecture	Sequential Time (seconds)	Parallelisation strategy	Parallel Time (seconds)	Speed-up
4-core Leon3	284.863	1	76.543	**3.7x**
		2	74.357	**3.8x**
NGMP GR740	89.599	1	25.222	**3.55x**
		2	24.178	**3.7x**

In both processor architectures, the two strategies present a very similar performance speed-up due to the small number of cores available in the architecture. This is not the case of the results presented in deliverable D1.1 (Figures 6 an 9) in which the second strategy outperforms significantly the first (i.e. 11x and 7x respectively). The reason is because speed-ups shown in D1.1 consider a much more powerful processor architecture featuring 16 cores, which enables the second strategy to take fully benefit of the extra parallelism.

In case of the LEON3 and NGMP architectures instead, their four cores are already fully utilize with the first strategy, and so not getting much benefit if the level of parallelism increases.

4 Conclusions

This document evaluates the GNU-GCC implementation of the OpenMP tasking model, used to parallelised a pre-processing sampling application, on a space platform composed of the RTEMS SMP OS and two LEON-based multi-core architectures: a quad-core LEON3 and a NGMP GR740.

We demonstrate that the OpenMP tasking model is fully compatible with the latest version of RTEMS SMP (see Table 1) and the two LEON-based multi-core architectures, obtaining a performance speed-up of 3.8x and 3.7x respectively (being closed to the ideal speed-up, i.e. 4x) and so efficiently exploiting the parallel performance opportunities of both multi-core platforms.

Overall, we conclude that the OpenMP parallel programming model is a good candidate to be adopted in the space domain.

[31] http://www.gaisler.com/index.php/products/boards/gr-cpci-gr740

We would like to remark that the results presented in this deliverable were not planned for this ITI contract, going beyond of what was stated in the *Statement of Work* document. Moreover, the results presented in this document have increased targeted TRL of this contract, from 3 to 4.