SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D2.4 –Final report, including the set of reference applications ported to the TERAFLUX platform

Due date of deliverable: 31st March 2014
Actual submission: 19th may 2014

Start date of the project: January 1st, 2010 Duration: 51 months

Lead contractor for the deliverable: BSC

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Date	Author	Organization	Change History
1	**06.03.2014**	**Tomasz Patejko, Rosa M. Badia**	**BSC**	**Initial version**
2	**03.04.2014**	**Tomasz Patejko, Rosa M. Badia**	**BSC**	**First Version**
3	**10.04.2014**	**Tomasz Patejko, Rosa M. Badia**	**BSC**	**Peer reviewed version**
4	**07.05.2014**	**Roberto Giorgi**	**UNISI**	**Review**

Release Approval

Name	Role	Date
Tomasz Patejko	**Originator**	**06.03.2014**
Rosa M. Badia	**WP Leader**	**30.04.2014**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**10.04.2014**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of contributors to the writing of the document.

Rosa M. Badia, Tomasz Patejko,
Rahul Gayatri and Nacho Navarro
BSC
Daniel Goodman, Salman Khan, Behram Khan,
Paraskevas Yiapanis, Mikel Lujan and Ian Watson
University of Manchester

Feng Li and Albert Cohen
INRIA

Daniel Gracia, Sylvain Girbal
THALES

Executive Summary

This document is the fourth deliverable of WP2, Benchmarks and Applications. The objective of this work package is to understand the runtime behavior of applications in order to establish a guideline in the design of the other components of the computing system in TERAFLUX. As TERAFLUX explores the design of highly parallel tera-device systems, a key step in the project is to understand the fundamental requirements of highly parallel applications and their implications on all layers of a computing system that supports a data-flow programming and execution model – from the programming model itself, down to extensions to commodity architecture.

The deliverable describes the results of the fourth year of the project in task T2.3. The activities performed in task T2.3 relate to the porting of applications to the project programming models. The deliverable gives a final report on reference applications ported to TERAFLUX platform. The deliverable gives detailed explanation on applications ported in the fourth year. Additionally, the deliverable presents implementation of the translation scheme from StarSs to OpenStream programming model and reports on performance evaluation of translated applications.

Of special interest is the section 4.2, where implementations of the industrial applications in several programming models are discussed and its performance evaluated in real platforms and in the TERAFLUX platform with up to 1024 cores. This evaluation demonstrates that with the parallel processing the TERAFLUX platform is able to achieve very promising gains, showing its capability to compute in parallel the different bursts, especially as the number of bursts of the input increases.

1 Introduction

This is the fourth deliverable of WP2, Benchmarks and Applications. While during the first two years of the project the objective was to understand and characterize the behaviour of the applications, the last two years of the project focused on the porting of the project reference applications to the project programming models.

While during year 1 the partners participating in WP2 defined the characterization methodologies and metrics to be used in the project applications, in year 2 the project partners performed extended characterization of these applications. In year 3, project partners continued the porting of the project applications to the programming models considered in the project[1].

The table in next section contains the final list of reference applications ported to TERAFLUX [12] programming models; more than 30. This deliverable reports on the more interesting aspects of the applications' porting performed during the last reporting period. The reader is referred to previous deliverables of the WP2 (especially D2.3) where previous results on porting applications were reported.

1.1 Document structure

The document is organized as follows: Section 2Reference applications lists the reference application of the project. Section 2.1 gives an overview of a translation scheme between StarSs and OpenStream programming models. Section 4 reports the more interesting aspect of porting applications to the project programming models. Finally, Section 5 presents some conclusions.

1.2 Relation to other deliverables

This deliverable has relation with D2.2 and D2.3.

1.3 Activities referred by this deliverable

This deliverable refers to the activities performed task T2.3 during the fourth year of the project. the activities performed task T2.3 during the fourth year of the project.

[1] Although the porting was initially planned for years 3 and 4, this activity started earlier.

2 Reference applications

Deliverable D2.3 presented the list of reference applications to be ported to the project programming models. The applications are listed below, and for each of them there is the Status column that reports on the status of the applications.

The status can be:

- In progress: the partners are working on the porting of this application
- Available: the porting finished and the application is available in the project application repository.

Table 1 List of reference applications

Benchmark	**Responsible partner**	**Programming model**	**Status**
Matmul	BSC INRIA UCY UNIMAN	StarSs OpenStream DDM Scala + TM	Available Available Available Available
Radix Sort	INRIA	OMP	Available
Barnes-Hut	BSC	StarSs	Available
Cholesky	BSC UCY INRIA	StarSs DDM OpenStream	Available Available Available
Sparse LU	BSC INRIA UCY	StarSs OpenStream DDM	Available Available Available
FFT2D	BSC INRIA	StarSs OpenStream	Available Available
SPECFEM3D	BSC	StarSs	Available
N Queens	BSC	StarSs	Available
Lee's Routing (Labyrinth)	UNIMAN BSC INRIA UNIMAN + UCY	Scala +TM StarSs OpenStream DDM + TM	Available Available In progress Available
Kmeans	UNIMAN BSC	Scala + TM StarSs	Available Available
Ssca2	UNIMAN	Scala + TM	Available
STAMP – Vacation	UNIMAN	Scala + TM	Available
Travelling Salesman	UNIMAN	Scala + TM	Available
MapReduce - wordcount	UNIMAN	Scala + TM	Available
MapReduce sorting	UNIMAN	Scala + TM	Available
Pregel – Pagerank	UNIMAN	Scala + TM	Available

Benchmark	**Responsible partner**	**Programming model**	**Status**
Pregel – Single Source Shortest Path	UNIMAN	Scala + TM	Available
FFT 1D	INRIA BSC	OpenStream StarSs	Available Available
Fmradio	INRIA BSC	OpenStream StarSs	Available Available
Picture-in-picture	INRIA INRIA	OpenStream Heptagon [11]	Available Available
Ad-hoc software radio	INRIA INRIA	OpenStream Heptagon	Available Available
Conv2d	UCY	DDM	Available
IDCT	UCY	DDM	Available
Trapez	UCY	DDM	Available
Graph 500	BSC UD	StarSs Codelet	Available Available
Go (Montecarlo Tree Search)	UNIMAN	Scala + TM	Available
Flux (object tracking)	BSC	StarSs	Available
GROMACS	BSC	StarSs	Available
PEPC	BSC	StarSs	Available
WRF	BSC	StarSs	Available
STAP (Radar)	Thales BSC INRIA	Seq. code StarSs	Available Available Available
Viola & Jones (Pedestrian detection)	THALES INRIA	Seq. code OpenStream	Available Available
HPL Linpack	BSC	StarSs	Available

2.1 Addressing reviewers recommendation

In the reports of the third review, the following recommendation was given for the future work:

The reviewers also emphasize the importance of porting of at least three reference applications not characterized in Task 2.2 to the selected programming models in order to maintain the rigorous methodology that the Consortium agreed on in Task 2.3.

According to this recommendation, the consortium decided to focus their effort in the following three applications that were not characterized in task T2.2: FFT1D (see section 4.1.4), FMRadio (see section 4.1.1), and Graph500 (see section 4.1.2).

Additionally, the partners also agreed to port and do analysis on four applications that were previously characterized in task T2.2: Lee routing (Labyrinth, see section 4.1.3), Specfem3D (see section 4.1.5), STAP (Radar, see section 4.3.2) and Viola Jones (Pedestrian detection, see section 4.3.1). The reason is that these applications are significantly more relevant than others that the consortium may work and will allow to do relevant analysis and comparison, in order to publish the results.

3 Translation scheme between programming models: OpenStream and StarSs (INRIA, BSC)

Deliverable D2.3 presented a translation methodology from StarSs regions to OpenStream streaming constructs. In this section we report on progress on implementing the translation algorithm, and show initial performance results.

3.1 Introduction

The OpenStream compiler [2, 3] is an entry point to the TERAFLUX compilation toolchain: applications parallelized with TERAFLUX programming models need to be translated manually or automatically to code annotated with OpenStream directives. We use StarSs memory regions [1] as a case study for translation to OpenStream. OpenStream and StarSs have different features with regard to how data used for computation are represented and how data dependencies are handled:

- OpenStream's basic unit of computation is a dataflow stream whereas StarSs applications use dynamic memory regions specified by the programmer for communication between tasks
- OpenStream requires explicit task dependencies to maintain correctness of parallel execution whereas data dependencies of StarSs tasks are inferred at runtime

Translation from StarSs code to OpenStream requires the programmer to identify data dependencies between StarSs tasks and to encode them with OpenStream streaming constructs. [2] introduces a methodology of how StarSs-OpenStream translation can be carried out at compile time. The idea behind automatic StarSs-OpenStream translation is to encode StarSs memory regions as a set of streams that contain versions of memory locations accessed by tasks. The most recent versions in the set of streams are calculated by a modified StarSs dependence resolver, and determine live data identified by the StarSs memory regions at the given point of application execution. The set of streams is attached to each OpenStream task, and is used by OpenStream runtime to determine data dependencies between tasks and to synchronize concurrent memory accesses.

3.2 Implementation

We have developed a source-to-source translator that carries out StarSs-OpenStream translation at compile time. Our implementation translates code that uses OmpSs [9], a programming model that extends OpenMP with features from the StarSs programming model.

The key components of the translator are a StarSs pragma parser and an OpenStream code generator. The parser parses StarSs pragmas, and identifies memory regions and their directionalities further used to calculate their live versions. The code generator generates calls to the StarSs dependence resolver and OpenStream task pragmas with the set of version streams. Generated code is further passed to the OpenStream compiler.

Figure 1 and Figure 2 show translated OpenStream code with corresponding StarSs programming constructs: `task` pragma with regions and `taskwait` pragma. When translator encounters StarSs `task` pragma (Figure 1), it extracts information about the regions and their respective directionalities, and generates the appropriate code (OpenStream code in lines 2-14). The code declares streams and initializes region descriptors with collected information, which will be used by the dependence resolver at runtime to handle data dependencies; a generated call to the dependence resolver is shown in line 15 in Figure 1. The generator also generates OpenStream `task` pragmas so the body of the

StarSs code

```
1. #pragma omp task in(in_a[0:SIZE-1], in_b[0:SIZE-1])\
                 out(out_c[0:SIZE-1])
2. void MatrixMultiply(float *in_a, float *in_b, float *out_c)
   {
3.   //Matrix multiplication code
   }
```

Generated OpenStream code

```
1. void MatrixMultiply (float *in_a, float *in_b, float *out_c)
   {
     //declaration of version streams
2.   int streams_peek[MAX_CONNECTIONS]__attribute__((stream_ref));
3.   int num_peek;
4.   int streams_in[MAX_CONNECTIONS] __attribute__((stream_ref));
5.   int num_in;
6.   int streams_out[MAX_CONNECTIONS] __attribute__((stream_ref));
7.   int num_out;
     //Information about regions and directionalities
8.   region_descriptor_t reg_desc[3];
9.   reg_desc[0].type = INPUT;
10.  reg_desc[0].id = in_a;
11.  reg_desc[1].type = INPUT;
12.  reg_desc[1].id = in_b;
13.  reg_desc[2].type = OUTPUT;
14.  reg_desc[2].id = out_c;

     //Call to OpenStream dependence resolver
15.  resolve_dependences(reg_desc, 3,
                         &streams_peek[0], &num_peek,
                         &streams_in[0], &num_in,
                         &streams_out[0], &num_out);
16.  int peek_view[num_peek][1];
17.  int in_view[num_in][1];
18.  int out_view[num_out][1];
     //OpenStream task pragma
19.  #pragma omp task peek(streams_peek >> peek_view[num_peek][0])\
                      input(streams_in >> in_view[num_in][1])\
                      output(streams_out << out_view[num_out][1])
     {
20.    //Matrix multiplication code
     }
   }
```

Figure 1 Translation of StarSs task pragma

task could be run in parallel when the generated OpenStream code is executed (lines 19 and 20).

On occurrences of StarSs `taskwait` pragma (StarSs code in Figure 2), the translator generates calls to two OpenStream functions that flush and restore correct version streams (lines 1 and 3, respectively, in OpenStream code in Figure 2). These two functions ensure correct synchronization

StarSs code

```
1. #pragma omp taskwait
```

Generated OpenStream code

```
1. openstream_resolver_flush_versions();
2. #pragma omp taskwait
3. openstream_resolver_restore_versions();
```

Figure 2 Translation of StarSs taskwait pragma

between OpenStream producers and consumers when the barrier is encountered.

Currently the StarSs-OpenStream translator handles the following StarSs pragma use cases:

- Function definitions
- C compound statements
- Function declarations with external linkage

3.3 Evaluation

The current prototype uses the dependence resolver that compares regions' start addresses, which makes it unusable for applications with tasks that access overlapping regions. The benchmarks were executed on a single node with 2 Intel E5649 (6-Core, 12M cache, 2.53 GHz) processors.

We used two benchmarks to evaluate performance of the translated applications: Cholesky and Matrix Multiplication. Both benchmarks were parallelized with the OmpSs programming model. In the OmpSs implementations tasks access blocks of data to perform local computations. The tasks are synchronized by data dependencies among pieces of memory expressed as OmpSs task regions, which do not overlap. Codes translated to OpenStream use OpenStream runtime for task scheduling and dependency resolver based on GNU C tree. Figure 3 shows speedup for different number of tasks for both benchmarks executed for 12 threads. Both benchmarks attain from 6-fold to 11-fold speedup for increasing number of tasks OmpSs and OpenStream runtimes need to handle.

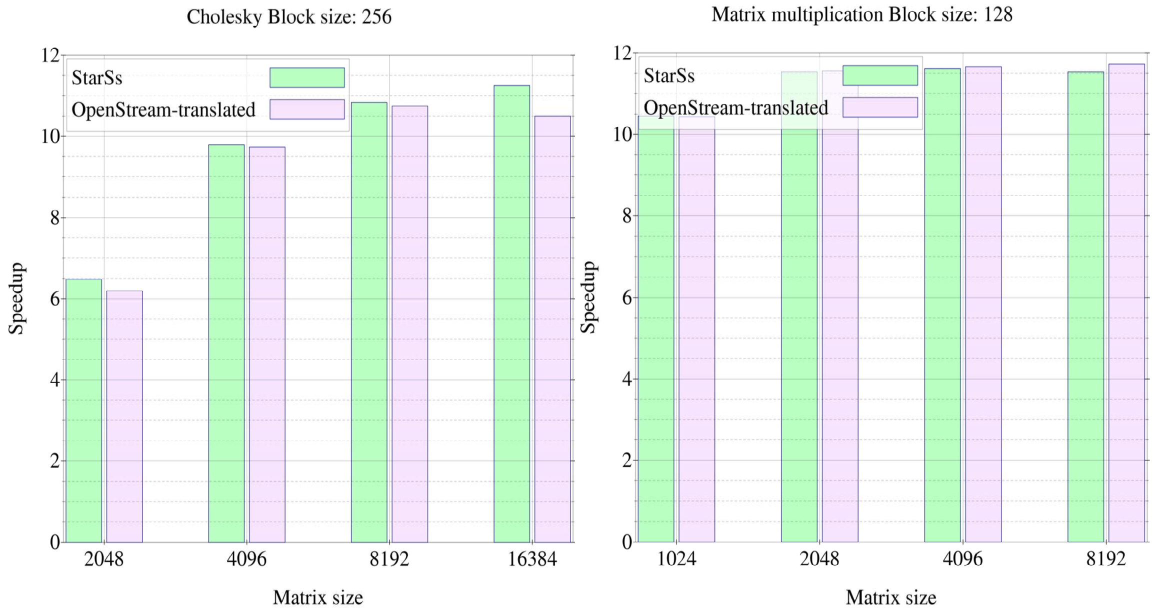


Figure 3 Task scalability for Matrix Multiplication and Cholesky benchmarks

4 Application porting

4.1 Applications ported to StarSs (BSC)

4.1.1 FMRadio

In the fourth year of the project we implemented a version of the parallel FMRadio application using the OmpSs programming model [9, 10]. In this section we give an overview on the parallel implementation of the application, and report on performance evaluation and analysis.

The sequential version of FMRadio application is composed of a set of filters that perform computations over the signal samples. The filters are executed sequentially in a loop iterating of the stream of samples. The main application loop of sequential FMRadio is shown in Figure 4.

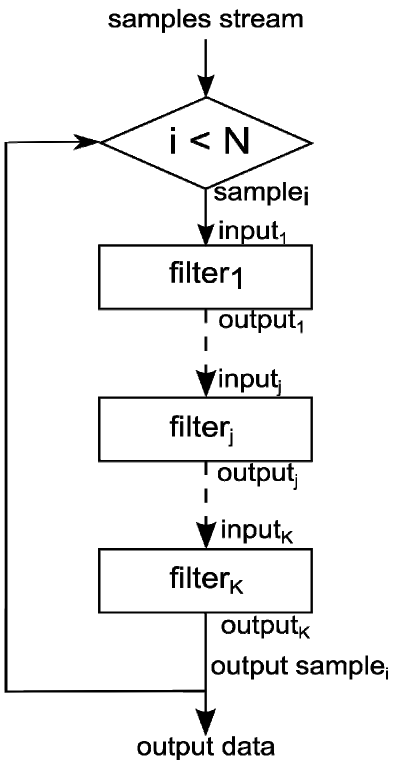


Figure 4 Body of the main loop of sequential FMRadio application

The iteration of the main loop starts by passing sample *i* to the first filter as its input data. Each filter *j* iterates over its input data *j* and performs calculations that result in output *j* that is passed further to the next filter *j+1* as an input *j+1*. The algorithm resembles a pipeline where each filter *j* works on the data produced by filter *j-1*, and passes the output to filter *j+1*. In addition, each filters maintains the history of its previous computations *i-l*, *1 < l < N*, that is needed for its execution *i*.

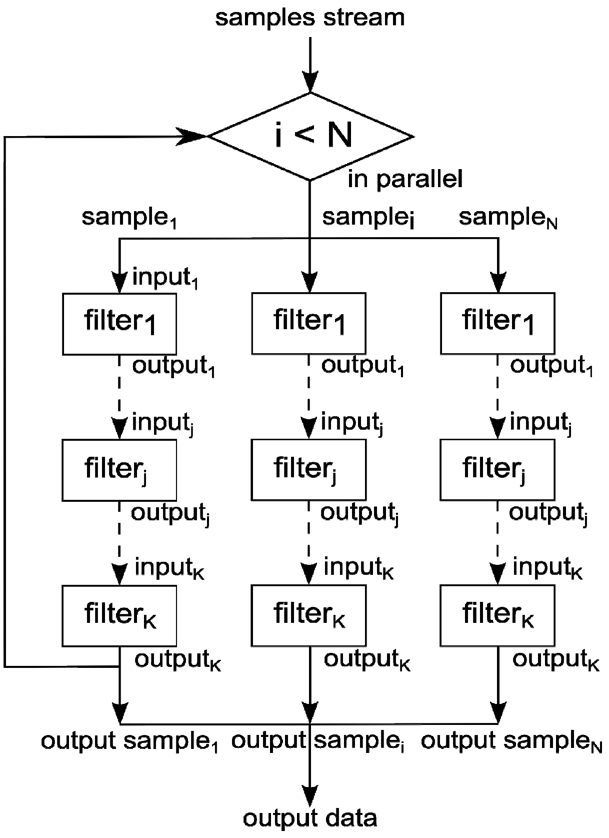


Figure 5 Body of the main loop of parallel FMRadio application

The parallel main loop of the application is shown in Figure 5. The parallelization strategy assumes encapsulating each filter *j* into an OmpSs task that can be run in parallel. Task synchronization is maintained through flow dependence between tasks *j-1* and *j*. Each set of filters executed in iteration *i* runs in parallel and performs local computations that do not depend on computations from iteration *i-1*. Such parallelization scheme requires no loop-carried dependencies between tasks. The computation histories maintained by the filters introduce such dependencies and in practice serialize the execution of tasks. The dependencies on the computation histories are relaxed by precomputing the necessary data before the main loop starts executing and by redundantly calculating the histories by tasks at each iteration of the main loop.

Figure 6 shows the speedup results for executions of FMRadio for varying grain sizes. The grain size is a parameter that is passed to the application and it indicates a size of the samples each filter will process. By varying the grain size values we can change the size of OmpSs tasks. The measurements were taken for grain size values ranging from 2 to 12. The application was run on a node with two processors Intel SandyBridge-EP E5-2670/1600 (20M cache, 8-core, 2.6 GHz) and memory 8x4GB DDR3-1600 DIMMS (2GB/core). The best speedup achieved by parallel FMRadio is 13-fold for grains of sizes 8 and 10 for 16 threads. As can be seen in the figure, excellent results can be obtained for grains between 8 and 10.

An image from an execution trace that shows tasks' execution appears in Figure 7. We can see that tasks from different FMRadio's main loop iterations run in parallel, whereas tasks processing the *i*th sample are serialized by flow dependence on the results of their computations.

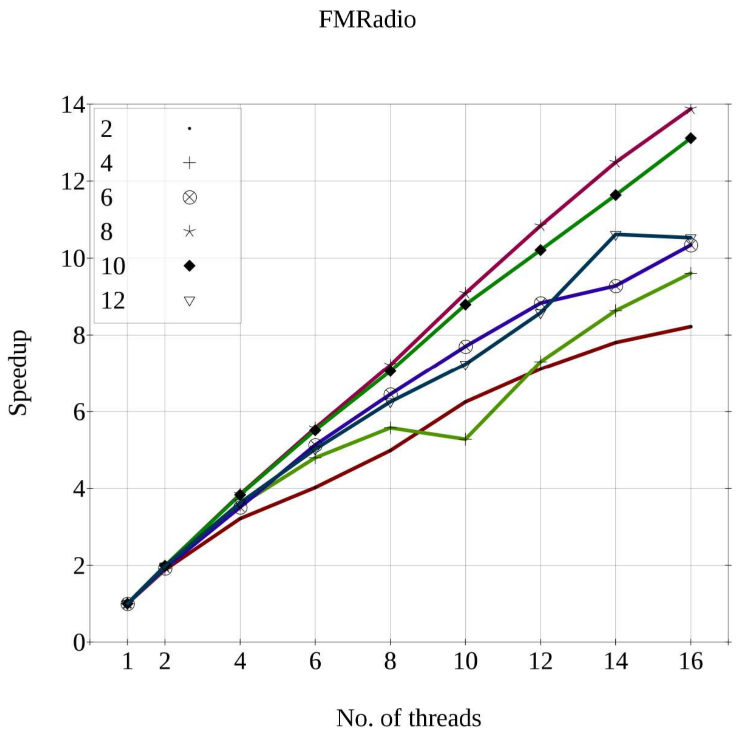


Figure 6 FMRadio speedup for different grain sizes

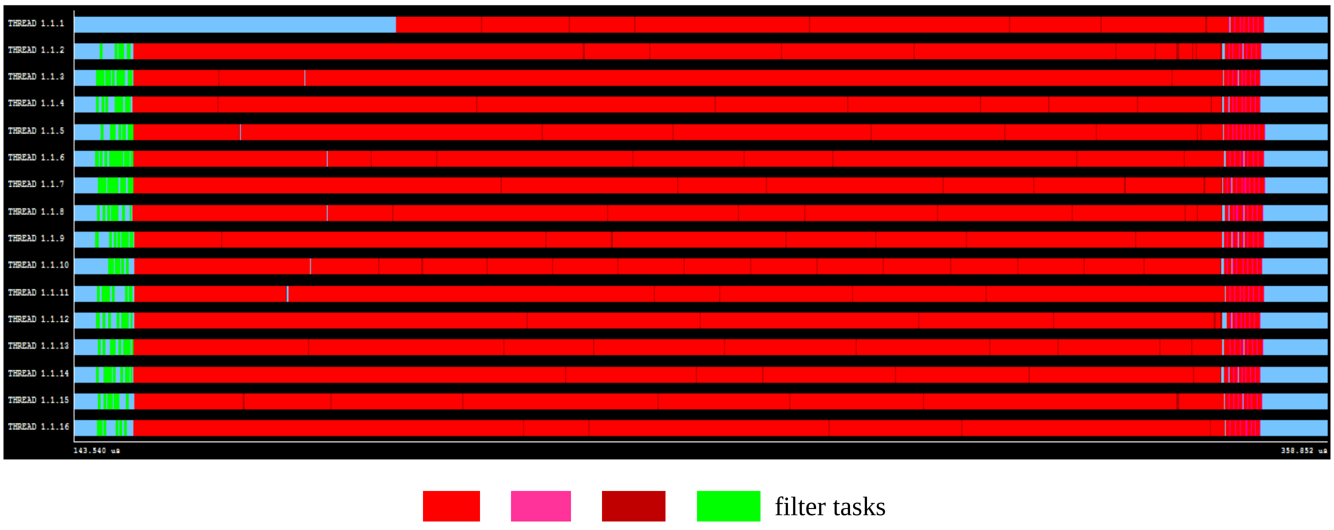


Figure 7 Trace with task execution of parallel FMRadio for grain size 8 for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

4.1.2 Graph500 – graph search problem

In the year four of the project we started working on the parallel implementation of the Graph500 [4] benchmark that uses OmpSs programming model as a parallelization framework. In this section we give an overview on the algorithms we implemented and present initial performance results.

Graph500 is a benchmark that implements Breadth First Search of a graph resulting in a spanning tree. It comes with a set of parallel implementations based on e.g. MPI and OpenMP. The graph can be represented as an adjacency list or in a Compressed Spare Row (CSR) format. The benchmark can be configured to use a Recursive Model for Graph Mining (R-MAT) or a Kronecker product as its graph generator.

In our experiments we use the Kronecker method to generate graphs, which are stored in CSR format. The algorithms we show in this section are Array-Read-Based and Queue-Based [5].

```
1.  def queue_bfs(graph, root) =
2.      current_frontier, next_frontier
3.      local_queues(threads_number)
4.      visited_nodes
5.      next_frontier.push(root)
6.      visiter_nodes(root) = VISITED
7.      while !next_frontier.empty?
8.          for nodes <- current_frontier.partition(threads_number)
9.              #pragma omp task
10.                 lq = local_queues(my_thread_id)
11.                 for cn <- nodes
12.                     for n <- graph.neighbors(cn)
13.                         if !visited_nodes.visited?(n)
14.                             if visited_nodes.atomic_cas(n, NOT_VISITED, VISITED)
15.                                 lq.push(n)
16.                                 if lq.full?
17.                                     next_frontier.sync_push(lq)
18.                                     lq.clear
19.                 if !lq.empty?
20.                     next_frontier.sync_push(lq)
21.                     lq.clear
22.             #pragma omp taskwait
23.             swap(next_frontier, current_frontier)
24.             next_frontier.clear
```

Figure 8 Pseudocode of Queue-based parallel Breadth First Search algorithm

The pseudocode of a Queue-based BFS algorithm is shown in Figure 8. The algorithm uses two queues, `current_frontier` and `next_frontier`, which store, respectively, nodes on the level *i* and nodes on the level *i+1* of the graph traversal. The constructed spanning tree is stored in a data structure called `visited_nodes`. The traversal continues until there are no nodes to visit (line 7). The set of nodes on the currently visited level *i* is partitioned among threads (line 8) and each thread runs a task that visits neighbors of an assigned subset of nodes (lines 9-21). The threads maintain local queues stored in a data structure called `local_queues`, where neighbors of nodes visited on level *i* are stored. The neighboring nodes will form a frontier on the level *i+1*. Nodes in the thread's local queue are moved to the `next_frontier` queue at the end of traversal (lines 19-21) or when the thread's local queue is full (lines 16-18 and 19-21).

Tasks do not maintain data dependencies. Although they access in parallel the `visited_nodes` data structure to mark nodes that have been visited, they are synchronized through test-test-and-set pattern (lines 13 and 14). Also, in the implementation accesses to a queue `current_frontier` are synchronized. Each thread modifies the queue's tail using atomic operation `fetch-and-add`. The `local_queues` queues are modified locally by each thread so there is no need to manage data dependencies for these data structures.

```
1.  def array_read_bfs(graph, root) =
2.      visited_nodes
3.      levels(graph.nodes_number)
4.      threads_completed(threads_number)
5.      visited_nodes(root) = VISITED
6.      level = 1
7.      levels(root) = level
8.      finished = false
9.      while !finished
10.         for nodes <- graph.nodes.partition(threads_number)
11.             #pragma omp task
12.                 for n <- nodes
13.                     if levels(n) != level
14.                         continue
15.                     for ng <- graph.neighbors(n)
16.                         if visited_nodes.visited?(ng)
17.                             if visited_nodes.atomic_cas(ng, NOT_VISITED, VISITED)
18.                                 levels(ng) = level + 1
19.                                 threads_completed(my_thread_id) = false
20.             #pragma task wait
21.             finished = threads_completed.logical_and
22.             level++
```

Figure 9 Pseudocode of Array Read-based parallel Breadth First Search algorithm

The pseudocode of Array Read-based BFS is shown on Figure 9. As it was a case for Queue-based BFS, the algorithm uses the `visited_nodes` data structure. It defines the `levels` data structure, which stores levels of nodes that were visited. The nodes on the level *i* form a current frontier (line 13 and 22). The nodes of the graph are partitioned and distributed among threads (line 10). Each thread runs an OmpSs task that checks whether a node forms a current frontier (line 13). Then it visits neighbors of the node (lines 15-19). If the neighbouring node has not been previously encountered, it is marked as visited and will form a frontier on the level *i+1* (line 16-19). The algorithm runs until there are no nodes to visit (lines 9, 19 and 21).

The Array Read-based BFS uses the `levels` data structure that stores current frontier. The data structure is partitioned among threads and allows local computations without creating data dependencies between tasks. So the algorithm does not introduce an overhead of concurrent accesses to queues with current and next frontiers present in Queue-based BFS.

Figure 10 shows performance results for OmpSs Queue-based and Array Read-based implementations of Graph500. We compared our results with sequential and OpenMP implementations. The application was run on a node with two processors Intel SandyBridge-EP E5-2670/1600 (20MB cache, 8-core, 2.6GHz) and memory 8x4GB DDR3-1600 DIMMS (2GB/core). The size of a graph is depicted by scale; the number of vertices in a graph is 2^{scale}. The performance metric used in the measurements is the number of traversed edges per second (TEPS). Due to memory limitations of our environment we could only run the application for scale of up to 22.

BFS is an irregular parallel algorithm and makes an interesting case for work distribution among tasks. At the early and late levels of traversal tasks are very small due to the small size of a frontier and the low number of nodes to visit, which are of small degree. There is no overlap of task creation with task execution; very little parallelism is available. In the middle levels of traversal the frontier of the BFS is larger, and the number of nodes to visit increases. Tasks become bigger and can be scheduled and executed in parallel.

Although OmpSs implementations are competitive with OpenMP implementations, for graphs used in our experiment there is not enough available parallelism due to uneven work distribution. Threads stay idle waiting for OmpSs tasks to execute.

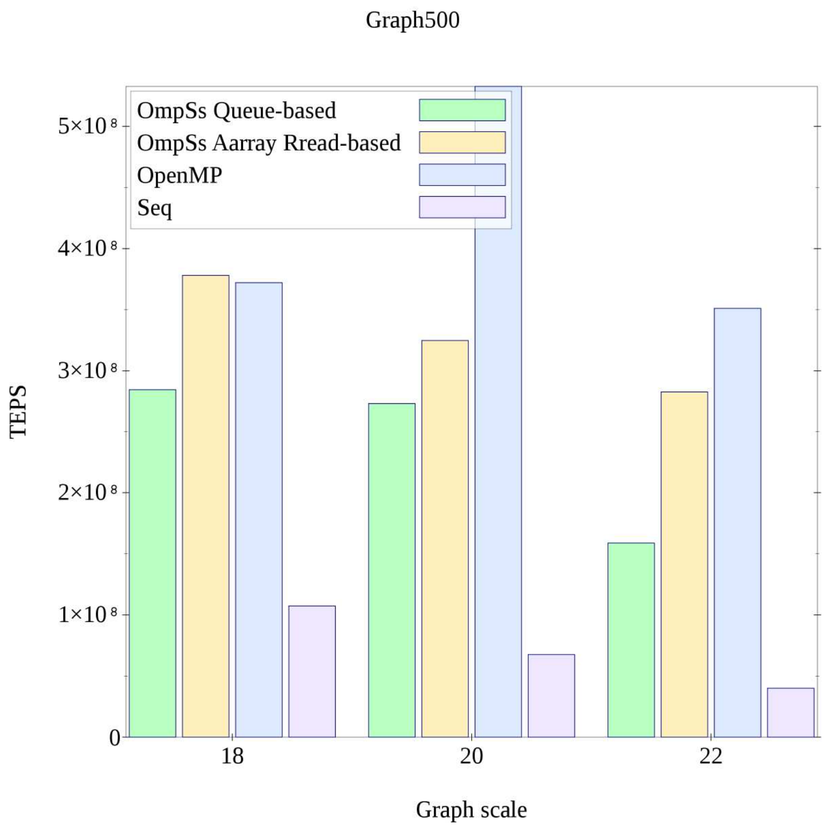


Figure 10 Performance results of Graph500 for different graph sizes

4.1.3 Labyrinth (Lee-Routing)

In deliverable D2.3 we presented a methodology for decomposing sequential code into parallel OmpSs tasks with Tareador [10]. As a running example we used Lee-routing algorithm. In this deliverable we present further performance improvements of the algorithm.

As it was stated in deliverable D2.3, a main grid is divided recursively into smaller subgrids; each task creates two subtasks that try to route paths within their respective subgrids. The algorithm takes a divide-and-conquer approach in order to route given set of paths between points on the board. In the current incarnation of the algorithm each subgrid is expressed as an OmpSs region of the main grid. OmpSs regions allow us to relax dependencies on the main grid and limit them to its parts where tasks perform routing.

Figure 11 shows pseudocode of expansion and traceback phases. The code for halving main grid and spawning children tasks has not been changed and we omit it.

```
1.  #pragma omp task in(grid) in(work_list) in(grid)
2.  // Compute paths
3.   def router_task(grid, work_list) =
4.  //Divide grid into subgrids and
5.  //create two children tasks that will route paths within the subgrids
6.  //…
7.          path_grid_list = allocate_list
8.          for((src, dest) <- my_work_list))
9.              expansion_queue = allocate_queue
10.             local_grid = copy_grid(grid)
11. #pragma omp task concurrent(path_grid_list) in(grid[lowerY:uppperY][lowerX:upperX])
12. //expansion task
13.                 if(expand(local_grid, expansion_queue, src, dest))
14. #pragma omp critical
15.                     list_insert(path_grid_list, (src, dest))
16. #pragma omp task out(grid[lowerY:upperY][lowerX:upperX]) concurrent(path_grid_list)
17. //traceback task
18.         for((src, dest) <- path_grid_list)
19.             path = traceback(grid, src, dest)
20.             if(not_empty?(path))
21.                 update(grid, path)
```

Figure 11 Pseudocode of Labyrinth (Lee routing) algorithm with OmpSs regions

Each expansion task (lines 11-15) computes a path on a part of the main grid. A subgrid expansion tasks work on is encoded as an OmpSs region of the main grid (`in` clause in task pragma, line 11). Whenever expansion for a given pair of points is successful, the pair is inserted into the `pair_grid_list` data structure (line 14), that is further used by the traceback task. Dependencies on `pair_grid_list` between expansion tasks are relaxed through OmpSs `concurrent` pragma, and the accesses to the list are synchronized by the OpenMP `critical` pragma (line 14).

The traceback task (lines 16-21) marks successfully expanded paths and it accesses only a part of the main grid. The updates on the grid are limited to the subgrid expansion has been previously performed on, so there is no need for synchronization on the entire grid. The dependence on a subgrid is expressed as an OmpSs `out` clause with the subgrid encoded as an OmpSs region (line 16). The same subgrid forms an antidependence between expansion tasks and a traceback task that synchronizes execution of these tasks. Furthermore, as expansion and traceback tasks are synchronized through dependences on a subgrid, there is no need to synchronize accesses to `path_grid_list` between instances of these two tasks.

The speedup results are shown in Figure 12. The application was run on a node with two processors Intel SandyBridge-EP E5-2670/1600 (20MB cache, 8-core, 2.6 GHz) and memory 8x4GB DDR3-1600 DIMMS (2GB/core). The application was executed for two data sets: board of size 512x512 and with 512 paths to route, and board of size 1024x1024 with 4096 paths. Measurements for both data sizes were calculated against sequential execution of the application.

The algorithm scales for varying number of data sets. It achieves 10-fold speedup for board of size 512x512 with 512 paths to route, and 8-fold speedup for board of size 1024x1024 with 4096 path to route.

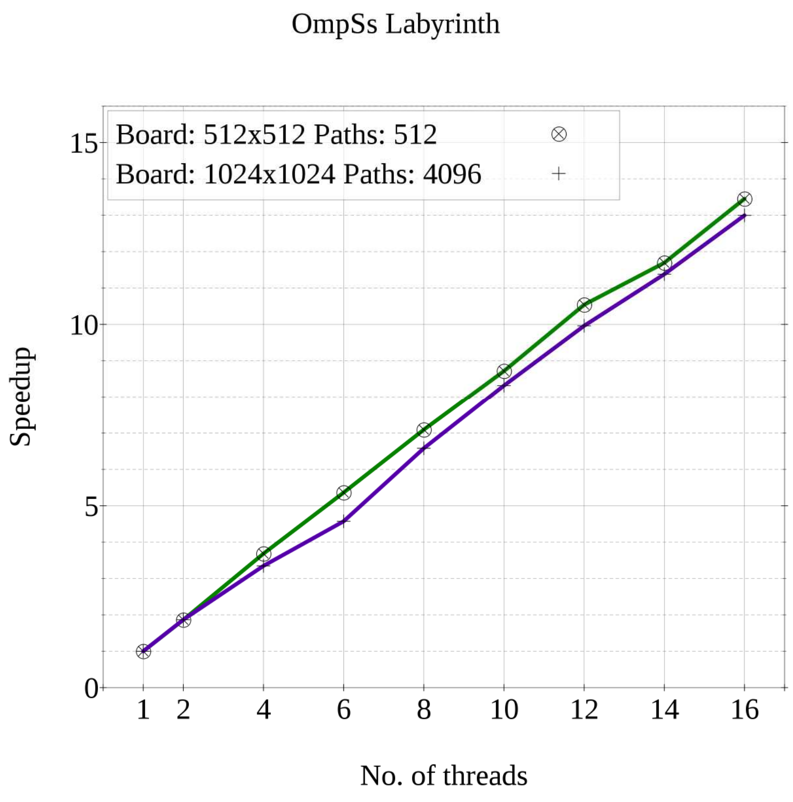


Figure 12 Speedup results for OmpSs Labyrinth for varying data sizes

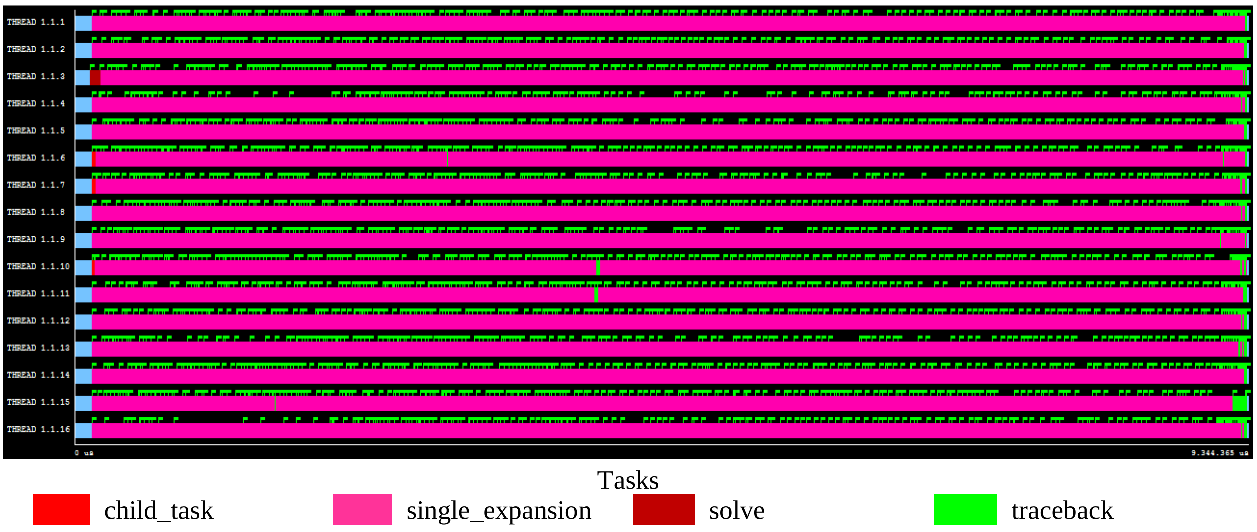


Figure 14 Trace with parallel task execution for board of size 1024x1024 and number of paths 4096 for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

Figure 14 shows a timeline with task execution for 16 threads. `Single_expansion` tasks, that perform expansion between points on the board, run in parallel. They maintain input dependencies between each other on respective subgrids, and their accesses to `pair_grid_list` data structure are handled by OpenMP `critical` pragma. `Traceback` tasks execute after the expansion tasks finished. Synchronization is maintained by antidependence on the subgrid expansion and traceback are performed on. Dependencies between expansion and traceback tasks can be seen in

🟥 child_task 🟥 single_expansion 🟥 solve 🟩 traceback

Figure 13. The lines in the figure indicate dependencies between tasks. We can see that no

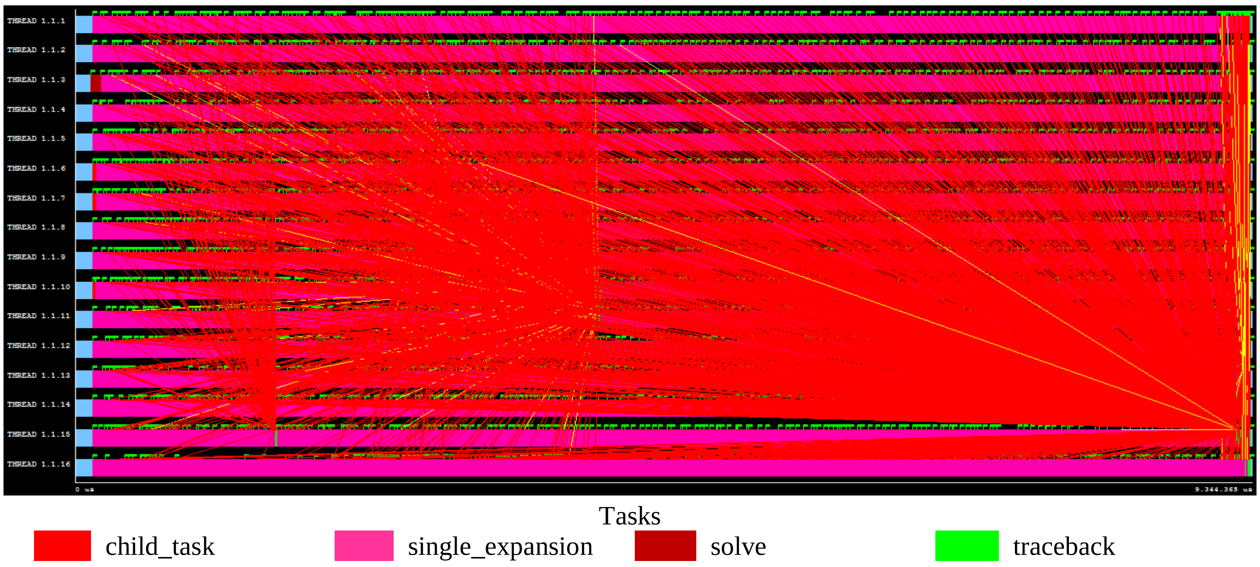


Figure 13 Execution trace with task dependencies for a board of size 1024x1024 and number of paths 4096 for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

dependencies between expansion tasks are found by the OmpSs runtime. The only dependencies that need to be resolved during execution are those between expansion and traceback tasks.

4.1.4 FFT1D – numeric application

In the year four of the project we used OmpSs as a parallelization framework for FFT1D benchmark. The implementation is based on the parallel algorithm that uses SMPSs [8] for parallelization. In this section we give an overview on the algorithm and present results of performance measurements for the code using OmpSs.

FFT1D implements the Fast Fourier Transform over an array of complex double precision floating point numbers and it uses 6-step Fast Fourier Transform algorithm [6]. The input and output data are unidimensional, but the algorithm operates over them as a bidimensional matrix. The 6 steps are divided into 3 transposition operations, 2 FFT operations over all rows and a multiplication of all the elements by twiddle factors. The benchmark uses FFTW library for FFT operations.

The pseudo-code of the algorithm is shown in Figure 15. Each step of the algorithm is implemented as an OmpSs task. Each tasks accesses blocks of matrix `A`, which are represented as OmpSs regions. Representing data as an OmpSs region allows OmpSs tasks running in parallel to overlap computations; data dependencies between tasks that access overlapping regions are tracked and resolved by the OmpSs runtime. Tasks `transpose_block` and `transpose_swap` (line 4, 6, 20 and 22) implement an in-place matrix transpose operation and task `fft1d` (lines 9 and 17) calls FFTW library to perform in-place FFT operation.

```
1.  void fft(complex A[N_SQRT][N_SQRT])
2.      // 1. Transpose
3.      for (long i = 0; i < N_SQRT; i += TR_BS)
4.          transpose_block(A[i][i])
5.          for (long j= i + TR_BS; j < N_SQRT; j += TR_BS)
6.              transpose_swap(A[i][j], A[j][i])
7.      // 2. First FFT round
8.      for (long j = 0; j < N_SQRT; j += FFT_BS)
9.          fft1d(A[j][0])
10.     // 3 & 4. Twiddle and Transpose
11.     for (long I = 0; I < N_SQRT; I += TR_BS)
12.         twiddle_transpose_block(i, A[i][i])
13.         for (long j = I + TR_BS; j < N_SQRT; j +=TR_BS)
14.             twiddle_transpose_swap(i, j, A[i][j], A[j][i])
15.     // 5. Second FFT round
16.     for (long j = 0; j < N_SQRT; j += FFT_BS)
17.         fft1d(A[j][0]);
18.     // 6. Transpose
19.     for (long I = 0; I < N_SQRT; I += TR_BS)
20.         transpose_block(A[i][i])
21.         for (long j = I + TR_BS; j < N_SQRT; j += TR_BS)
22.             transpose_swap(A[i][j], A[j][i])
```

Figure 15 Pseudocode of the FFT1D algorithm

The speedup results of parallel FFT1D are shown in Figure 17. The application was run on a node with two processors Intel SandyBridge-EP E5-2670/1600 (20MB cache, 8-core, 2.6GHz) and memory 8x4GB DDR3-1600 DIMMS (2GB/core). The measurements were carried out for the following configurations:

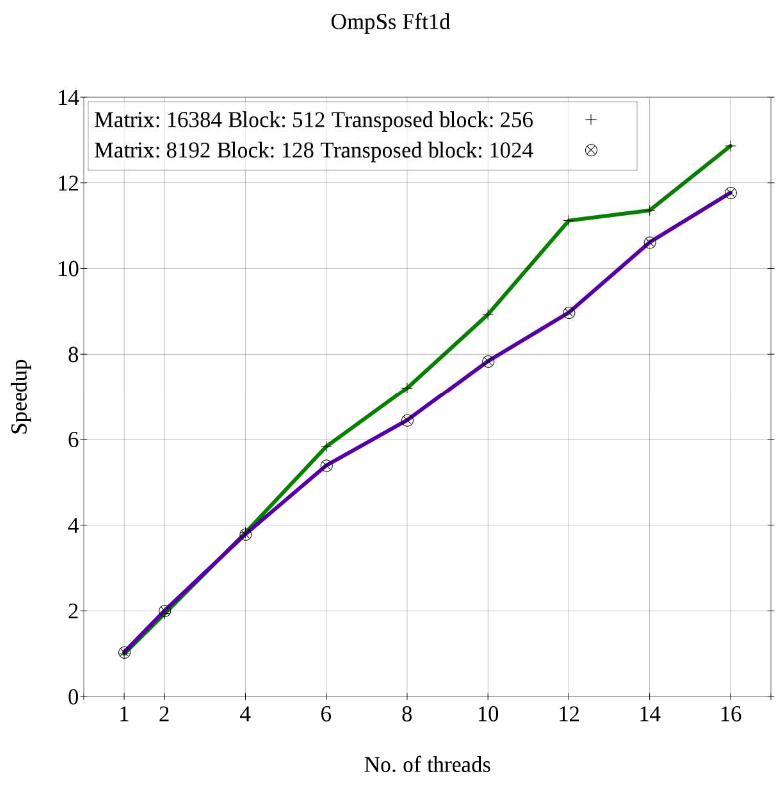


Figure 17 FFT1D speedup for different problem sizes

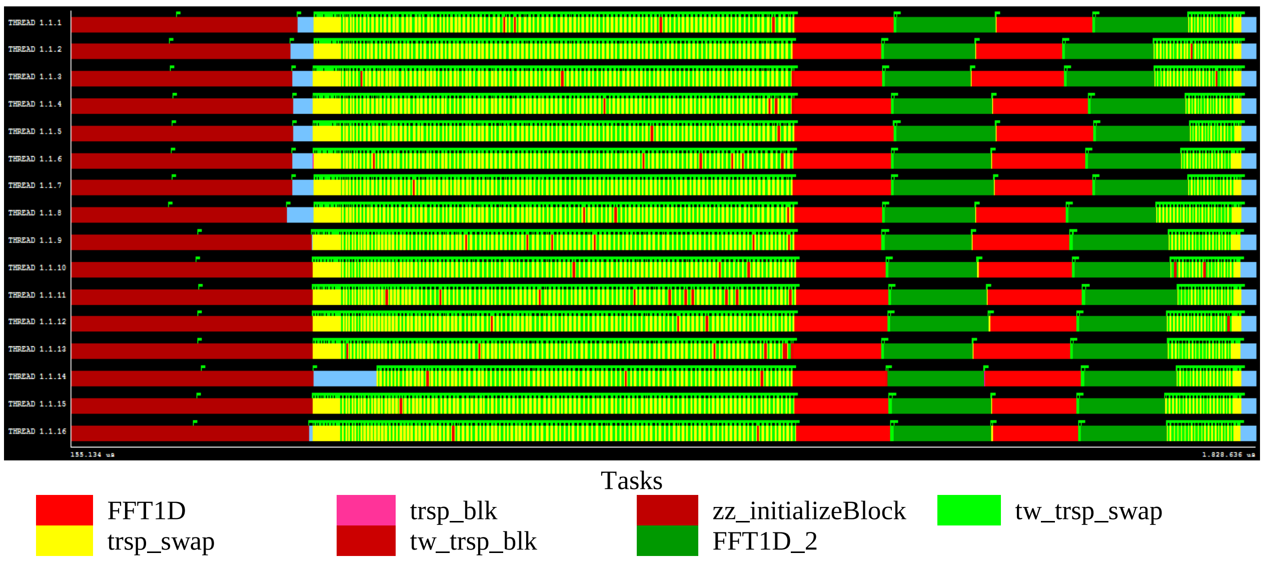


Figure 16 Trace with task execution of parallel FFT1D for the first configuration for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

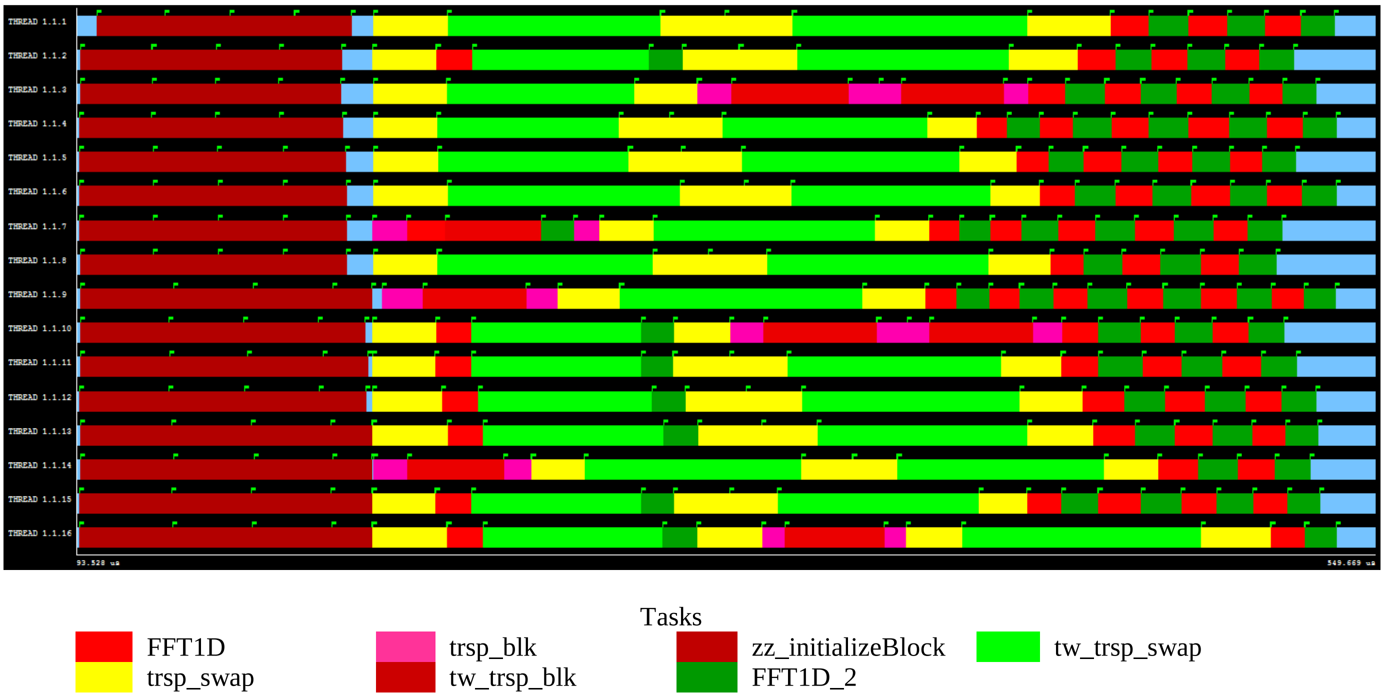


Figure 18 Trace with task execution of parallel FFT1D for the second configuration for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

1. 16384 elements, block size of 512 and transposed block size of 256
2. 8192 elements, block size of 128 and transposed block size of 1024

For both configurations the application scales well attaining 12-fold speedup in both cases against the sequential application.

Figure 16 and **Error! Reference source not found.** show parallel execution traces of FFT1D for both configurations. The traces show that FFT1D performs computations in steps. Tasks within each step are executed in parallel. In both cases the execution is dominated by tasks *trsp_swap* and *tw_trsp_swap*, which swap blocks of the matrix. Both tasks swap elements of the block in place and their size depends on the size of transposed blocks.

4.1.5 SPECFEM3D – scientific application

In the deliverable D2.2 we characterized CPI stack of SPECFEM3D benchmark. In the deliverable for the fourth year of the project we conveyed measurements and performance evaluation of the benchmark.

SPECFEM3D is a benchmark that implements spectral element method that is used for numerical calculations for 3-D wave propagation. Implementation that is presented in this section uses OmpSs for parallel execution.

As it was shown in deliverable D2.2, tasks covering most of the execution time are executed in the serial time loop in the following order:

1. `gather` – localizes and maps points in the local mesh into mesh elements from a global mesh

2. `process_element` – performs local calculations on a single 3D spectral element
3. `scatter` – sums computations on spectral elements into the global mesh

The first two tasks can be run in parallel as they locally perform accesses and calculations. The third task, `scatter`, updates a global mesh of points that can be shared among neighboring spectral elements, so the accesses to each point need to be serialized. OmpSs provides three clauses, which can be used within task pragma, that provide task serialization:

- directionality clauses (`in`, `out`, `inout`) – allow OmpSs to build task dependency graph at runtime and to schedule tasks based on their data dependencies
- `commutative` clause – allows OmpSs to execute tasks out-of-order in a serialized manner. The clause preserves data dependencies between tasks
- `concurrent` clause – relaxes data dependencies between tasks. The burden of providing correct task synchronization is put onto a programmer

Figure 20 shows SPECFEM3D speedup results for three implementations of `scatter` task that use the aforementioned methods of synchronization. The application was run on a node with two processors Intel SandyBridge-EP E5-2670/1600 (20M cache, 8-core, 2.6GHz) and memory 8x4GB DDR3-1600 DIMMS (2GB/core). *OmpSs-inout* and *OmpSs-conmmutative* implementations use, respectively, `inout` and `commutative` clauses. *OmpSs-concurrent* implements relaxed accesses through the `concurrent` clause; synchronized accesses are provided by OpenMP `atomic` pragma.

In all three cases application attains a 2-fold to 3-fold speedup for 16 threads. The difference in performance between concurrent implementation and the other two is attributed to synchronization overhead related to OpenMP `atomic` pragma. The `scatter` tasks of *OmpSs-concurrent* implementation cover more execution time compared to the tasks' codes of *OmpSs-inout* and *OmpSs-commutative* implementations (Figure 22, Figure 19, Figure 21). *OmpSs-inout*'s `scatter` tasks are scheduled onto the single core and in practice run sequentially, whereas *OmpSs-commutativ*'s `scatter` tasks are scheduled to and executed by different cores. The difference in scheduling policies for commutative tasks and tasks with `inout` clause comes from that inout tasks are executed in the order they were created; commutative tasks can be executed out of order.

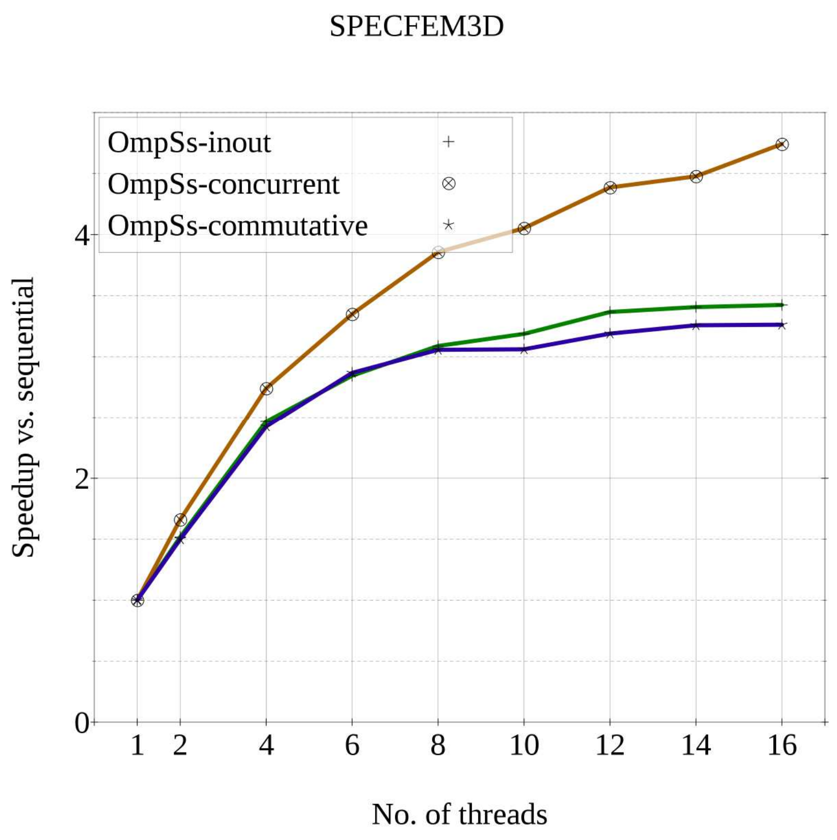


Figure 20 Speedup for different implementations of SPECFEM3D

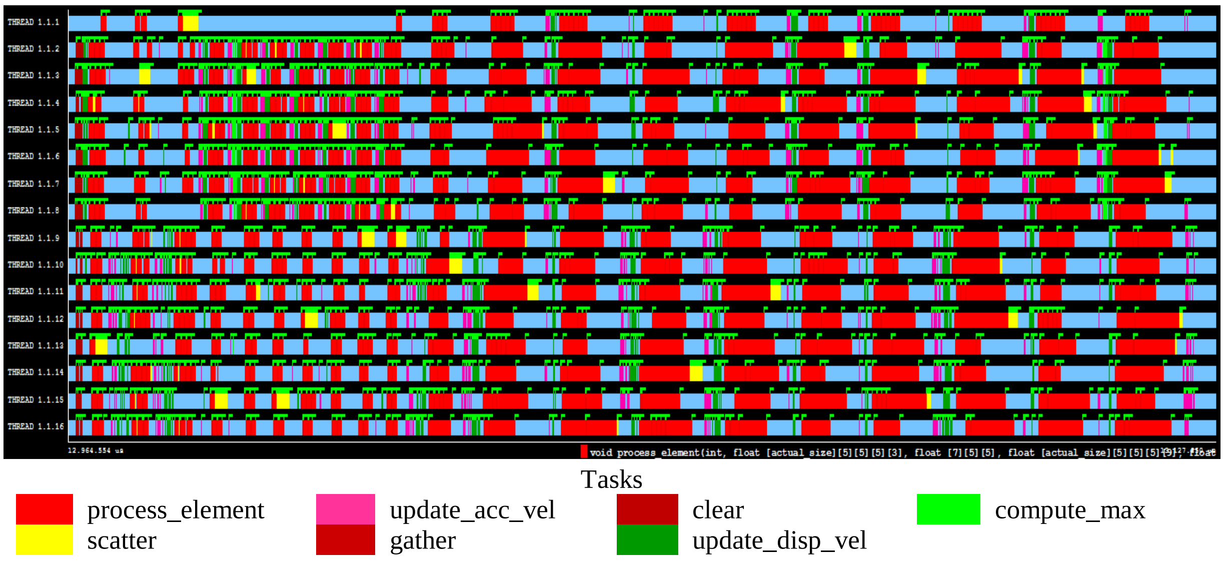


Figure 19 Trace with parallel task execution of OmpSs-inout implementation for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

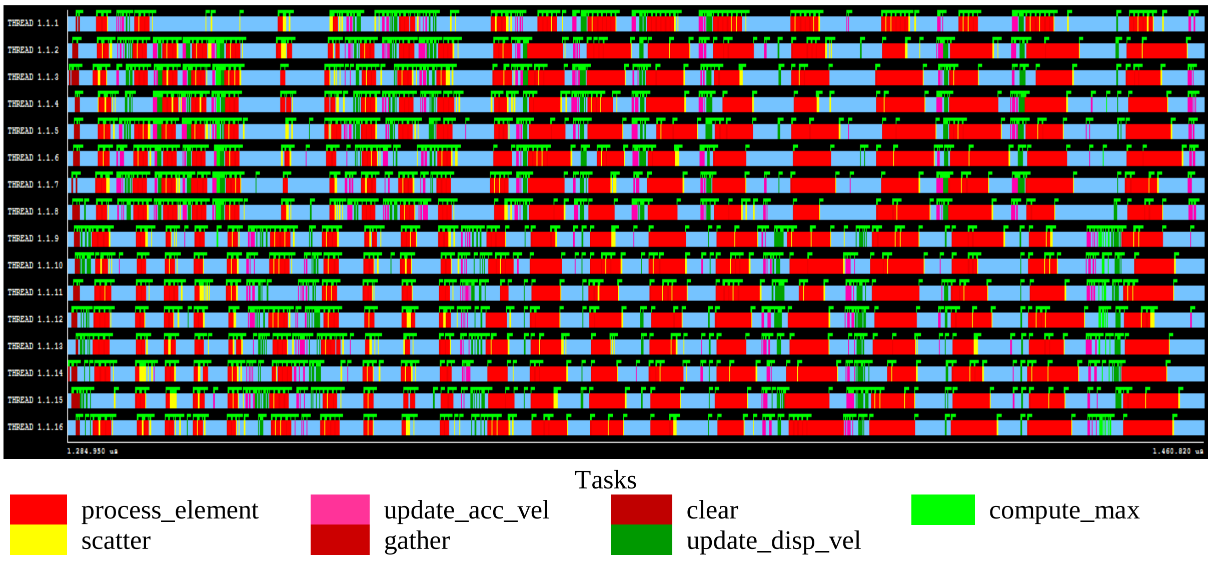


Figure 21 Trace with parallel task execution of OmpSs-commutative implementation for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

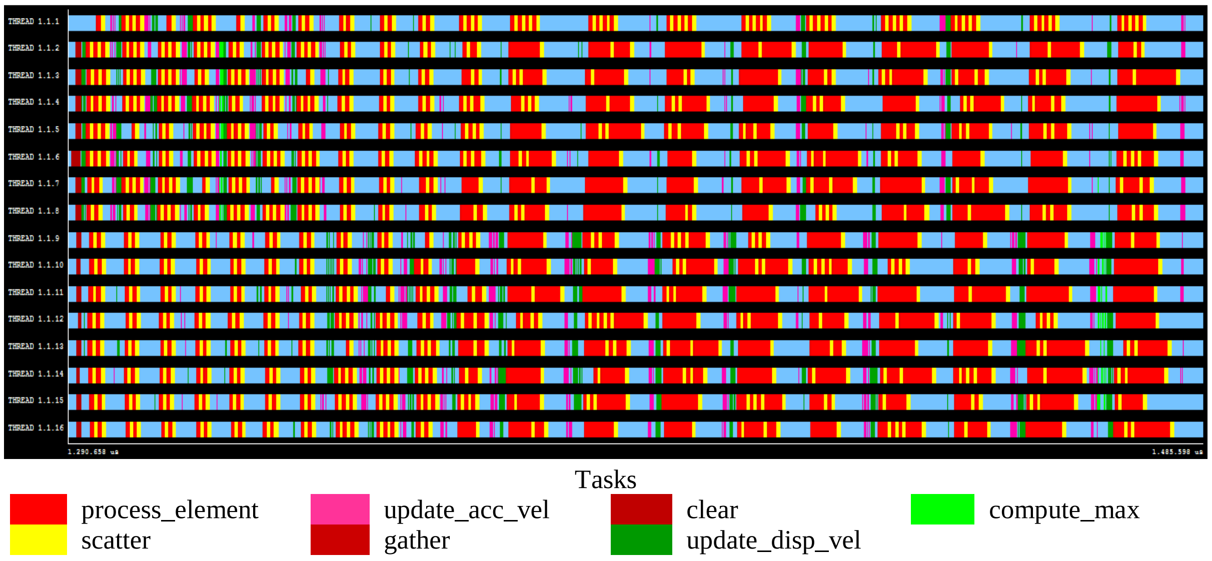


Figure 22 Trace with parallel task execution of OmpSs-concurrent implementation for 16 threads. Each row represents the activity of a thread, with different colors representing different tasks. The x-axis represents time.

4.2 Applications ported to Scala (UNIMAN)

This section highlights some of the benchmarks that have been ported to the Scala programming language. Within TERAFLUX we have developed two novel libraries which provide transactional memory and dataflow execution in Scala.

MUTS http://apt.cs.manchester.ac.uk/projects/TERAFLUX/MUTS/

DFScala http://apt.cs.manchester.ac.uk/projects/TERAFLUX/DFScala/

The DFScala library provides the functionality to construct and execute dataflow graphs. The nodes in the graph are dynamically constructed over the course of a program and each node executes a function which is passed as an argument. The arcs between nodes are all statically typed. An example of a function using DFScala and implementing Fibonacci follows:

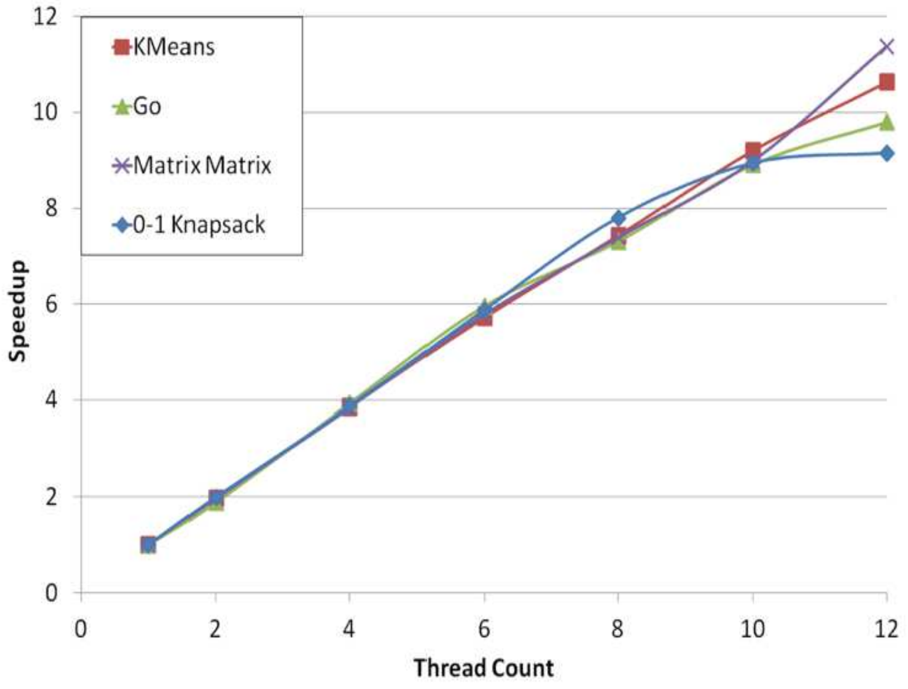
Expanded Version

```
        def fib(n :Int , out:Token[Int]){
                if(n <= 2)
                        out(1)
                else {
                        var t1 = DFManager.createThread(
                                (x:Int, y:Int, out:Token[Int]) => {out(x + y)}
                                )
                        var t2 = DFManager.createThread(fib _)
                        var t3 = DFManager.createThread(fib _)

                        t2.arg1 = n - 1
                        t2.arg2 = t1.token1
                        t3.arg1 = n - 2
                        t3.arg2 = t1.token2
                        t1.arg3 = out
                }
        }
```

Concise Version

```
        def fib(n :Int):Int = {
                if(n <= 2)
                        1
                else
                        fib(n-1) + fib(n-2)
}
```

The concise version can be used without having to change anything in Scala and it is clearly more productive. The applications that we are going to consider are Matrix-Matrix multiplication, 0-1 Knapsack, LeeTM, KMeans, Monte-Carlo Tree Search for playing Go, and Parallel Scala Collections. Matrix-Matrix multiplication and 0-1 Knapsack represent problems that can be solved without the need to use shared state. KMeans is drawn from the Stamp Benchmarks. The Go playing A.I. application represents a more dynamic environment and is a real world application. Go uses Monte Carlo Tree Search to determine the best move to play in a game of Go. This involves generating trees of combinations of moves for which a score is obtained and transactions are used to protect the shared state. The following graph presents the speedup results for these benchmarks on system with two 6 core AMD Opteron sockets.


KMeans
Go
Matrix Matrix
0-1 Knapsack
Speedup
Thread Count
0
2
4
6
8
10
12

4.3 Industrial applications (THALES with the collaboration of INRIA and BSC)

This section discusses the study of porting to the OpenStream and OmpSs dataflow programming paradigms of two applications used in THALES solutions and their evaluation on x86 multi-core solutions and the TERAFLUX platform.

The objective of THALES was threefold:

1. Evaluate the porting (complexity, effort required) of applications with parallelization opportunities to dataflow programming paradigms using OpenStream and OmpSs.
2. Evaluate the performances achieved by these solutions on commercial of-the-shelf (COTS) solutions (e.g., x86 multicores) when compared to more common programming parallelization solutions, like OpenMP.
3. Evaluate the performances achieved by the dataflow version on a massively parallel machine like the TERAFLUX platform with up to 1024 cores.

When porting, THALES wanted to take advantage of the capabilities of the parallelization solutions (OpenMP, OpenStream and OmpSs) to parallelize the applications, without discarding completely the legacy code. For that purpose we avoided extreme reorganizations of the legacy code or alternative algorithms for the same tasks, as the proposed programs already contained enough dataflow and non-dataflow parallelization options as discussed in previous deliverables (see D2.1, D2.2 and D2.3).

The results of this study are presented in the following sections. Sections 4.2.1 and 4.2.2 respectively present the porting of the Pedestrian Detection and Radar applications and the evaluation of the ports against OpenMP solution and on different machines (including the TERAFLUX platform). Finally, Section 4.2.3 discusses non-quantifiable metrics about the possible usage of the dataflow paradigms on industrial solutions.

4.3.1 The Pedestrian Detection application

The Pedestrian Detection application was previously described in D2.1, D2.2 and D2.3. It's based on

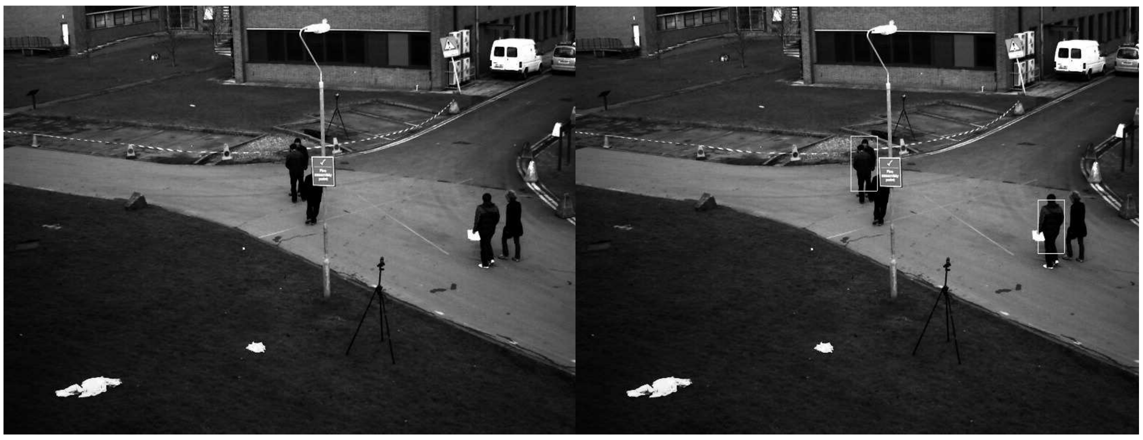


Figure 23 Example of output produced from the Pedestrian Detection application

the Viola-Jones cascading algorithm [7]. The application basically takes as input an image (possibly with people on it) and detects the people present on it. In addition it takes two other input parameters:

- The classifier with the filters used by the cascading algorithm implemented in the application to determine if there is a person in the input image.
- The scale step which determines the tiles scales to be analyzed. Typical scale ranges go from 1.01 to 1.2 in current real time implementations of the application.

The application outputs a copy of the input image with the detected people surrounded with white boxes. An example of the output generated from a given image, with a scale step of 1.01 and a basic classifier, can be seen in Figure 23. As can be observed not all the people from the input image are detected; the quality of the detection actually depends on the quality of the classifier and the scale applied.

Figure 24 shows the classical implementation of the cascading algorithm used for this study. In addition to the filtering of the tiles at the different scales with the cascading algorithm, the application performs a reduction on tiles belonging to the same scale for which detection was positive. Effectively as the tiles overlap each other the cascading algorithm tends to detect multiple tiles for the same person and the reduction allows merging them into a single detection.

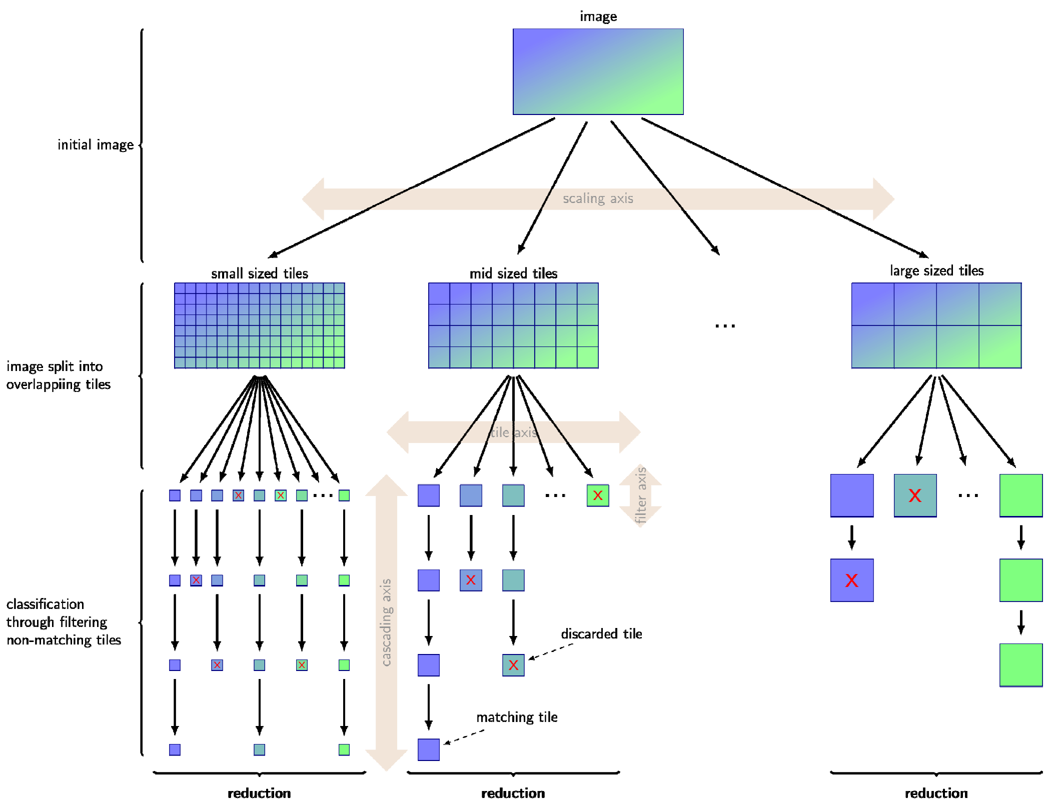


Figure 24 Cascading algorithm

4.3.1.1 Porting to parallel and dataflow paradigms

This section presents the efforts developed to achieve a parallel and dataflow implementation of the Pedestrian Detection application with good performance on x86 multicore machines and the TERAFLUX platform (ISA agnostic, but first evaluated on x86 cores). For this study two of the dataflow programming solutions were considered: OmpSs (BSC) and OpenStream (INRIA). One goal of the study was to achieve performance without a heavy modification of the initial application, by simply adding the parallelization and dataflow primitives of OmpSs and OpenStream.

A first thing to notice is that the Pedestrian Detection application presented few opportunities to apply dataflow principles at coarse level, i.e. between the different kernels that compose the application. Effectively the filtering task is done independently for each of the scales analyzed, and even the filtering of the different tiles on a scale is done independently. The only dataflow operation occurs when on a given scale the filtering operations have finished indicating the tiles on which a pedestrian was detected on a matrix, and the reduction task process the matrix to merge detected tiles. Thus, the only dataflow potential of the application was represented by this matrix, which we refer to as the detection matrix.

In D2.3 we presented four different directions for the parallelization of the application filtering task (scale, tile, filter and cascading axes in Figure 24). It is important to note that those cascading axes are typically implemented incrementally. Thus, when parallelizing at the tile, filter and cascading axes, the implementation also includes the scale axis parallelization. Our experiments demonstrated that parallelizing at these axes without parallelizing at the scale axis degraded the performance whereas speedup was achieved when using parallelization at the scale axis.

4.3.1.1.1 Parallelizing across the tile, the cascading and the filter axes

The tile axis parallelization proved to be very simple to implement by using the OmpSs and the OpenStream programming paradigms (as was also the case when using OpenMP). The code snippet in Figure 25 shows the tile axis parallelization applied to the Pedestrian Detection application when using OmpSs:

```
// process scale
for (irow = 0 ; irow < num_rows ; irow++) {
  for (icol = 0; icol < num_cols; icol++) {
// parallelize tile processing
#pragma omp task output (detectionMatrix[irow][icol])
    {
      good = runFilterStageOnTile(tile[irow][icol],0);
      for (istage = 1; good && istage < num_filter_stages; istage++) {
        good = runFilterStageOnTile(tile[irow][icol], istage);
      }
      detectionMatrix[irow][icol] = good;
    }
  }
}

#pragma omp task input ([irows][icols]detectionMatrix)
{
  // reduce task
}
```

Figure 25 The Pedestrian Detection application parallelized across tile axis with OmpSs

The OpenStream version was not much different, but differed on its dataflow approach and was more easily implemented. However when executing this version, we observed the creation of a large number of small threads. The first filters (i.e., filter at stage 0) discarded most of the tiles making most of the threads execute for a very short amount of time, causing a huge overhead, and thus degrading the performance.

Likewise, the parallelization of the cascading axis proved to be very straightforward as depicted by code snippet in Figure 26, but we observed important slowdowns when performing this parallelization:

```
// process scale
for (irow = 0 ; irow < num_rows ; irow++) {
  for (icol = 0; icol < num_cols; icol++) {
    for (istage = 0; istage < num_filter_stages; istage++) {
#pragma omp task output (detectionMatrix[istage][irow][icol])
      detectionMatrix[istage][irow][icol] =
          runFilterStageOnTile(tile[irow][icol], istage);
    }
  }
}


#pragma omp task input ([num_filter_stages][num_rows][num_cols]detectionMatrix)
{
  // reduce task
}
```

Figure 26 The Pedestrian Detection application parallelized across cascading axis with OmpSs

Effectively the cascading axis parallelization option executes all the filters for a given tile, increasing with another dimension (number of filter stages) the size of the streamed data (`detectionMatrix`) and leaving extra work to the reduction task. This approach proved to be worse than the tile parallelization because the amount of work on a filter was too small. Additionally it increased the size of the data being streamed.

Finally, for the filter axis parallelization approach we were forced to slightly modify our code and exploit some of the features of the OmpSs and OpenStream dataflow programming paradigms by streaming the `detectionMatrix` not only between the filter task and the reduction task, but also between the different filter stages as depicted in the code snippet in Figure 27.

The approach also provided worse performance than the sequential version, as it creates a large amount of threads that have little or no work to perform and that have to stream the `detectionMatrix` between each other.

```
// process scale
#pragma omp task output ([num_rows][num_cols]detectionMatrix)
for all irows & all icols → detectionMatrix[irows][icols] = true;

for (istage = 0; istage < num_filter_stages; istage++) {
#pragma omp task \
  input ([num_rows][num_cols]detectionMatrix) \
  output ([num_rows][num_cols]detectionMatrix)
  for (irow = 0 ; irow < num_rows ; irow++) {
    for (icol = 0; icol < num_cols; icol++) {
      if (detectionMatrix[irow][icol])
        detectionMatrix[irow][icol] =
          runFilterStageOnTile(tile[irow][icol], istage);
      else
        detectionMatrix[irow][icol] = false;
    }
  }
}

#pragma omp task input ([num_rows][num_cols]detectionMatrix)
{
  // reduce task
}
```

Figure 27 The Pedestrian Detection application parallelized across filter axis with OmpSs

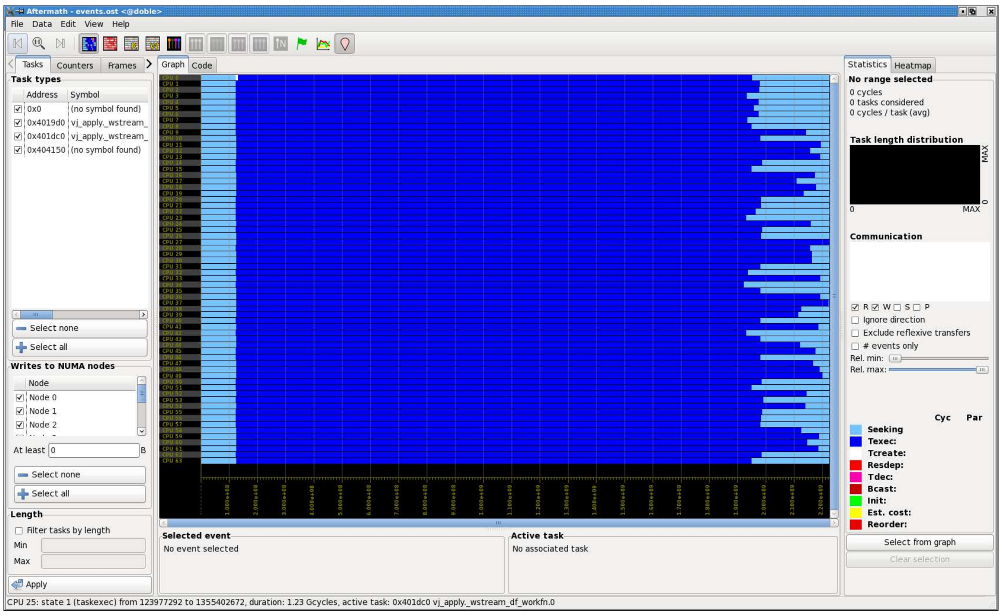
4.3.1.1.2 Parallelizing across the scale axis and the "balanced" scale axis

As for the parallelization across the cascading and tile axes, the parallelization across the scale axis proved to be straightforward as depicted in the code snippet in Figure 28:

```
…
#pragma omp task
foreach iscale on scales {
  // compute num_rows and num_cols on given scale
  // process scale

  // filter task
#pragma omp task output ([irows][icols]detectionMatrix[iscale])
  {
    for (irow = 0 ; irow < num_rows ; irow++) {
      for (icol = 0; icol < num_cols; icol++) {
        good = runFilterStageOnTile(tile[irow][icol],0);
        for (istage = 1; good && istage < num_filter_stages; istage++) {
          good = runFilterStageOnTile(tile[irow][icol], istage);
        }
        detectionMatrix[irow][icol] = good;
      }
    }
  }

  // reduce task
#pragma omp task input ([irows][icols]detectionMatrix[iscale])
  {
    // reduce task code
  }
}

#pragma omp taskwait
… // generate output image
```

Figure 28 The Pedestrian Detection application parallelized across scale axis

This implementation provided tasks with enough work to justify the creation of threads and, as expected, provided interesting speedups (see Section 4.2.1.2). However we observed that there were some cores that were unused before others as depicted in Figure 29.



Task types

	Address	Symbol
✓	0x0	(no symbol found)
✓	0x4019d0	vj_apply._wstream_
✓	0x401dc0	vj_apply._wstream_
✓	0x404150	(no symbol found)

Selected event
No event selected

Active task
No associated task

```
CPU 25: state 1 (taskexec) from 123977292 to 1355402672, duration: 1.23 Gcycles, active task: 0x401dc0 vj_apply._wstream_df_workfn.0
```

Figure 29 Trace of the parallelization across scale axis for the reference image

The reason for such unbalanced core usage was due to two different reasons:

1. In some of the scales more tiles are detected as positives making the filtering task more time consuming than others, as more filtering stages are executed.
2. Initial scales generate smaller tiles and thus more tiles to analyze. The effect is especially important when comparing initial scales with the last scales.

We have little room for action to resolve the first issue, as we cannot know a priori in a scale how many positives there will be. However, the second cause can be addressed by splitting the filtering task depending on the number of tiles to be analyzed. We call this approach the “balanced” scale axis parallelization approach, see bottom of Figure 30. The parallelization across the scale axis creates a thread for each scale, see mapping v0 in the figure; while the “balanced” parallelization creates a relatively large number of threads for the smallest scale (smaller tiles) and reduces the number of threads as the scale increases (larger tiles), see mapping v1 in the figure.

We implemented the “balanced” scale axis parallelization approach with both OpenStream and OmpSs, and observed that thanks to this approach all the cores presented a more homogeneous execution time usage on the reference image; see Figure 32. However, as we will see in Section 4.3.1.2 this proved to be a worthless solution when used over a large dataset.

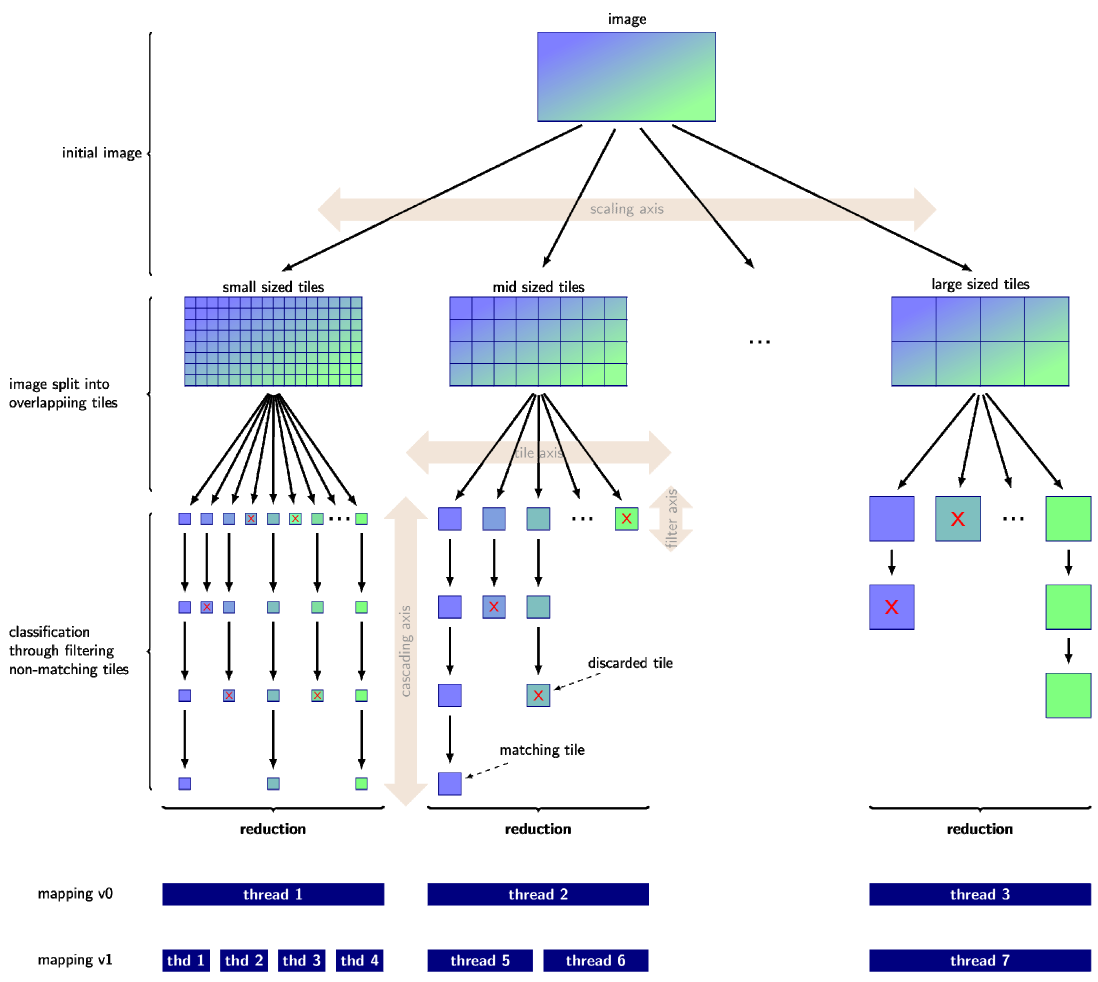


Figure 30 Scale axis vs. "balanced" scale axis parallelization approaches

4.3.1.2 Evaluation

This section presents the evaluation of the Pedestrian Detection application ports on different target architectures. The evaluation starts with the evaluation of the ports against current commercial off-the-shelf (COTS) multi-cores (Intel and AMD) and follows with an evaluation of the scalability of the parallelized application on the TERAFLUX platform using the OmpSs port with OWM.

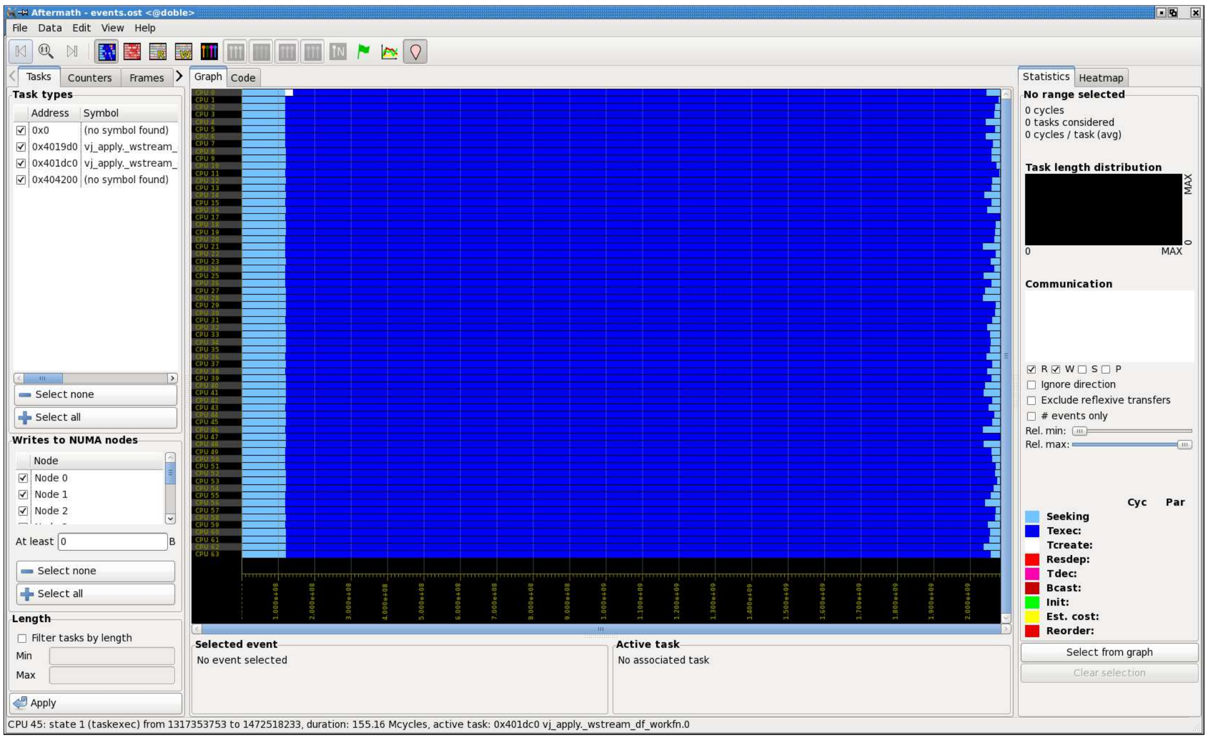
The Pedestrian Application was ported to OpenStream and OmpSs using the two parallelization options discussed in Section 4.3.1.1.2. For comparison purposes the application was also ported to OpenMP using the parallelization across the scale axis approach, and removing the dataflow operations by a `taskwait` pragma, as can be seen in the code snippet in Figure 31.

The five implementations were compared against the initial non-parallelized version of the application on 3 different target machines:

- Dual-core Intel i7 (2c in Figure 33)
- Four-core Intel i7 (4c in Figure 33)
- 16-core AMD Opteron (16c in Figure 33)

```
…
#pragma omp task
foreach iscale on scales {
  // compute num_rows and num_cols on given scale
  // process scale

  // filter task
#pragma omp task
  {
    for (irow = 0 ; irow < num_rows ; irow++) {
      for (icol = 0; icol < num_cols; icol++) {
        good = runFilterStageOnTile(tile[irow][icol],0);
        for (istage = 1; good && istage < num_filter_stages; istage++) {
          good = runFilterStageOnTile(tile[irow][icol], istage);
        }
        detectionMatrix[irow][icol] = good;
      }
    }
  }

#pragma omp taskwait

  // reduce task
  {
    // reduce task code
  }
}

#pragma omp taskwait
… // generate output image
```

Figure 31 The Pedestrian Detection application parallelized with OpenMP



	Address	Symbol
✓	0x0	(no symbol found)
✓	0x4019d0	vj_apply._wstream_
✓	0x401dc0	vj_apply._wstream_
✓	0x404200	(no symbol found)

No range selected
0 cycles
0 tasks considered
0 cycles / task (avg)

Selected event: No event selected
Active task: No associated task

CPU 45: state 1 (taskexec) from 1317353753 to 1472518233, duration: 155.16 Mcycles, active task: 0x401dc0 vj_apply._wstream_df_workfn.0

Figure 32 Trace of the “balanced” parallelization across scale axis for the reference image

Figure 33 shows the average performance speedup achieved with each of the ports on each of the platforms using as input 97 different images and 1.01 as scale step. As can be observed in the figure all the ports achieve a speedup that is close to the number of cores, just being slightly lower for all but the OpenStream balanced port in the 16-core configuration. There seems to be no gain on using the OpenStream or the OmpSs programming paradigms over OpenMP for the 2- and 4-core configurations, but the dataflow paradigms clearly achieve slightly better speedups on the 16-core machine, thus showing a better scalability as the number of processors increases. The balanced ports showed little speedup difference when compared to non-balanced ports. While there are input images that seem to take advantage from the balanced implementation, others show slowndowns. Additionally, the speedup varies depending on the number of cores. At the end the average speedups of balanced and non-balanced ports are very close and the balanced version presents little interest given the extra complexity they added in the source code. Finally, the OpenStream and the OmpSs versions seem to perform similarly and on par or better than more mature solutions as OpenMP.

On the TERAFLUX platform we evaluated the speedup scalability of the OpenStream with OWM (Owner Writeable Memory, cf. D7.1, D3.5) support port against a non-parallelized execution on the same platform. Only the parallelization across scale axis was considered, as the added complexity of the balanced solution was not providing any improvement in average. The OWM support was used to avoid the transmission of the large read-only inputs each of the tasks requires: the image and the classifier. Without the OWM support the threads would flood the interconnect with requests to retrieve the inputs, which would impair any speedup, especially as the number of threads increases. OWM avoids this flooding by creating local copies of those inputs. Various configurations of the TERAFLUX platform were considered, from 1 to 32 nodes of 32 cores, thus effectively simulating 32 to 1024 cores configurations. Figure 34 shows the average performance speedup achieved on each of the configurations using the default input and 1.01 as scale step. As can be observed the achieved speedup doesn't scale with the number of cores for the given inputs.

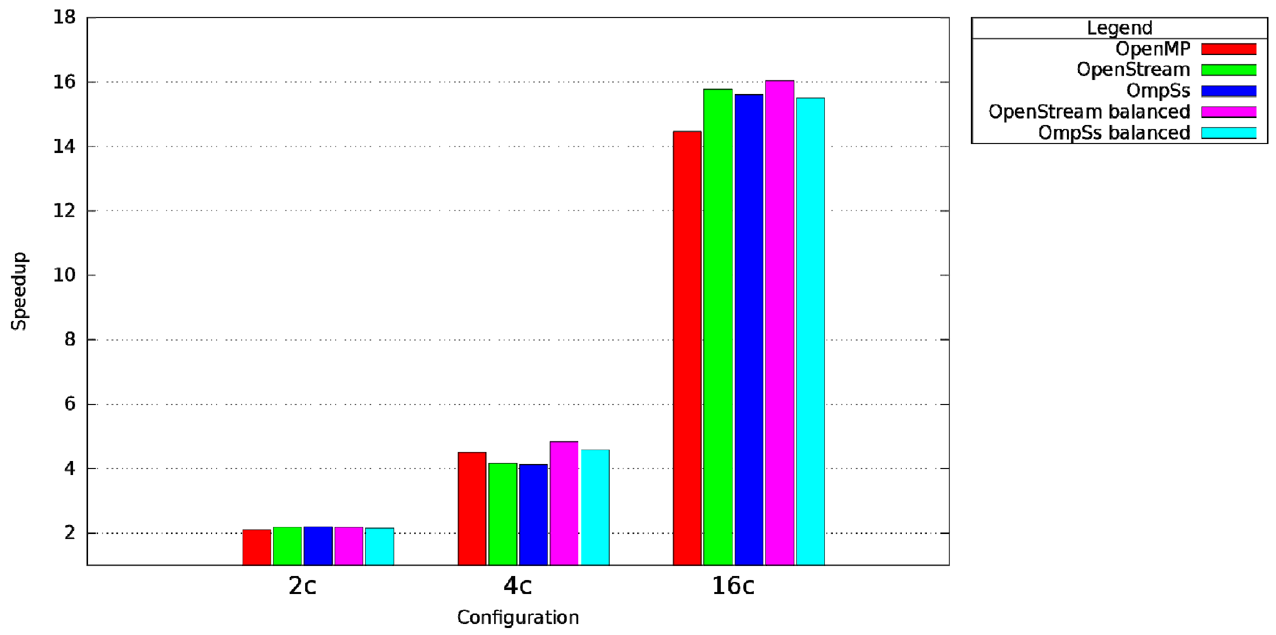


Figure 33 Ports speedup over COTS configurations

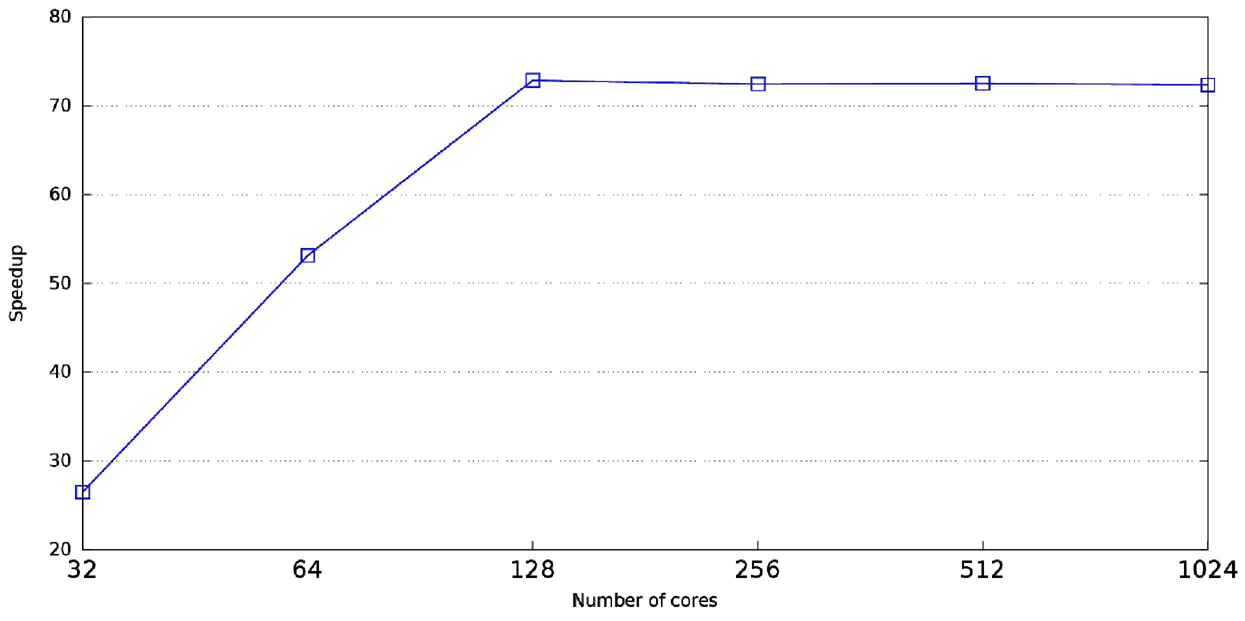


Figure 34 OpenStream port speedup over the TERAFLUX platform

The reason for the lack of scalability is the ability of the application to create enough threads that will run simultaneously. However the application is able to create more threads if it is given bigger images or if smaller scale step are provided. For example with a scale step of 1.008 we achieved a 111-fold speedup on a 128 cores TERAFLUX configuration. Likewise, with a scale step of 1.002 we achieved a 353-fold speedup on a 1024 cores TERAFLUX configuration. Table 2 shows some of the speedups achieved when using different scale step (bigger and smaller than 1.01) on different TERAFLUX configurations.

Table 2 Impact of the scale step parameter on performance scalability

Number of cores	Scale step	Speedup
32	1.03	24
32	1.02	26
64	1.02	37
64	1.015	47
128	1.015	49
128	1.012	61
128	1.008	111
1024	1.002	353

4.3.2 The Radar application

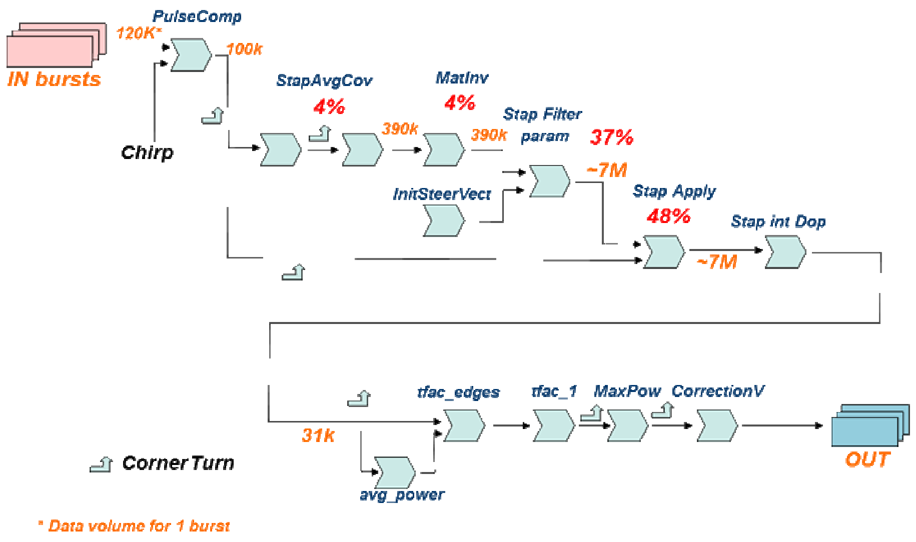


Figure 35 The Radar application kernel pipeline

As for the Pedestrian Detection application the Radar application was introduced and discussed in D2.1, D2.2, and D2.3. The Radar application is a pure data flow application based on a Space-Time Adaptative Processing (STAP) signal processing technique. It is typically implemented on planes taking input bursts from the radar device processing the bursts in a pipeline fashion and producing as output the objects in movement that have been detected, see Figure 35. Additionally there is no state shared between burst, thus no data dependencies exist other than those in the pipeline.

4.3.2.1 Porting to parallel and data flow paradigms

Unlike for the Pedestrian Detection application the kernel of the Radar application shows a large number of opportunities to apply the data flow programming paradigms provided by OmpSs and OpenStream. However, simply applying the data flow programming on the whole kernel may create an unbalanced solution where the amount of data transferred generated by the different tasks on the pipeline (see Figure 35) outweighs any gain that may be achieved by the implemented parallelism and pipelining.

For that purpose we used the Paraver (for the OmpSs porting) and the Aftermath (for the OpenStream) performance debugging tools to evaluate performance gain of our implementations. We initiated the porting task by applying the OmpSs and OpenStream data flow pragmas between all the tasks (including the transposition operations) that can be observed in Figure 35. The resulting application presented an important slowdown when compared against the sequential implementation.

Figure 36 shows the output of the performance debugging tools over the kernel of the initial porting of the Radar application. Each line represents one core; in this case the kernel is being executed in a 16 core machine. The white sections display the creation of the kernel tasks, the red sections represent tasks using most of its execution time spent to perform data transfers, the dark (vivid) blue sections represent tasks using most of its execution time performing computations, and finally the light blue sections represent time when the cores are idle. As can be observed most of the computation tasks are very small and most of the time is spent by data transfers and the cores remain idle during large time in between the computation tasks. The large data transfers are actually caused by the large amount of computation tasks created.

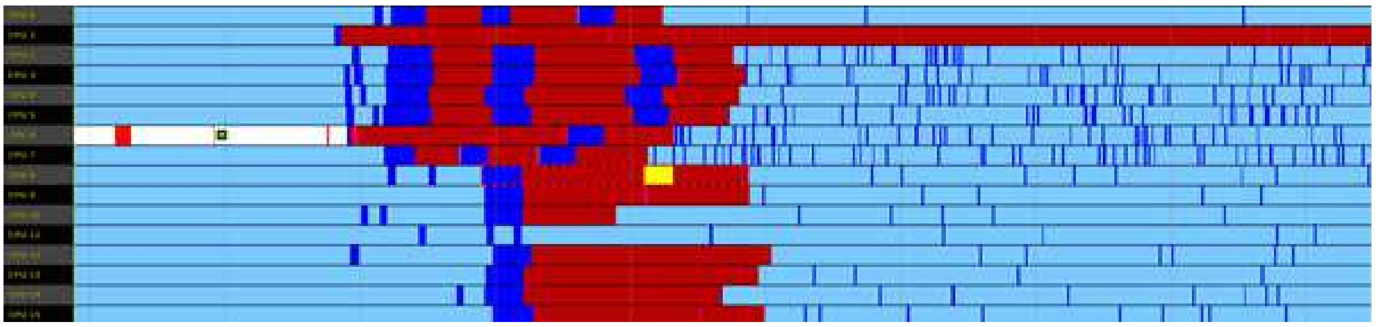


Figure 36 Performance of non-debugged Radar application

We iterated over this initial implementation by grouping tasks, maximizing computation time over idle and data transfer time. Figure 37 presents the final result of the performance debugging task over the Radar application kernel (the color code is the same than the one for Figure 36). As can be observed the computation time has been maximized, the data transfer time minimized (they cannot be observed in the screenshot but they are there), and the idle time almost eliminated. The performance measurements of this implementation will be presented in the following section.

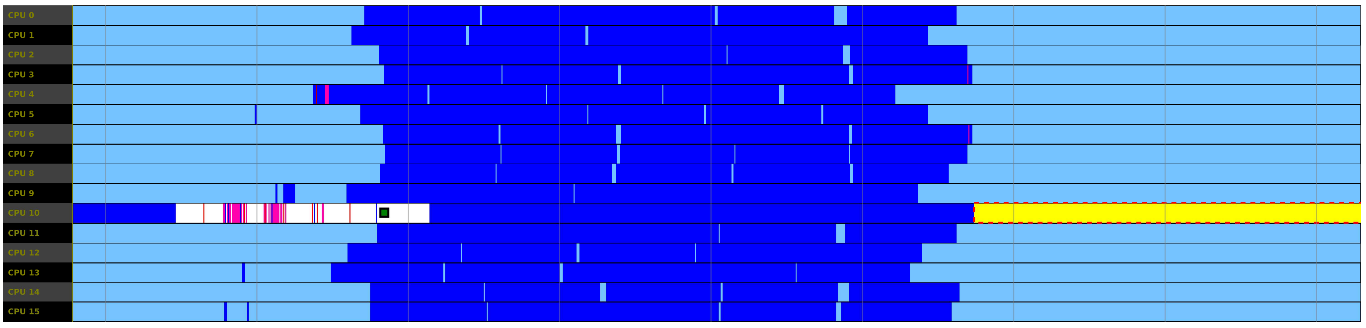


Figure 37 Performance of debugged Radar application

4.3.2.2 Evaluation

As for the Pedestrian Detection application the ported Radar application was evaluated on four different machines:

- Dual core Intel i7 (2c in the following figures)
- Four core Intel i7 (4c in the following figures)
- 16 core AMD Opteron (16c in the following figures)
- TERAFLUX platform (with different configurations ranging from 32 cores to 1024)

Three different inputs were used during the evaluation:

- small: consisting of a trace of three bursts of a radar signal (~800KB)
- large: consisting of a trace of nine bursts of a radar signal (~2.6MB)

- huge: consisting of a trace of seventy five bursts of a radar signal (~21MB)

Finally two different implementations of the Radar application were evaluated:

- Sequential implementation processes the bursts sequentially, but each burst is using the parallel kernel.
- Parallel implementation processes all the inputs bursts in parallel, each of the bursts using the parallel kernel.

The Radar application was also parallelized using the OpenMP solution to compare it against the parallel data flow OmpSs and OpenStream implementations.

Figure 38 and Figure 40 show the performance gains achieved by the parallel implementations on the Intel and AMD platforms when using the sequential and parallel bursts processing implementations. As can be observed the sequential processing of the bursts suffers from the initialization and finalization phases of the kernel, impeding performance gains. The OmpSs version seems to handle better these phases.

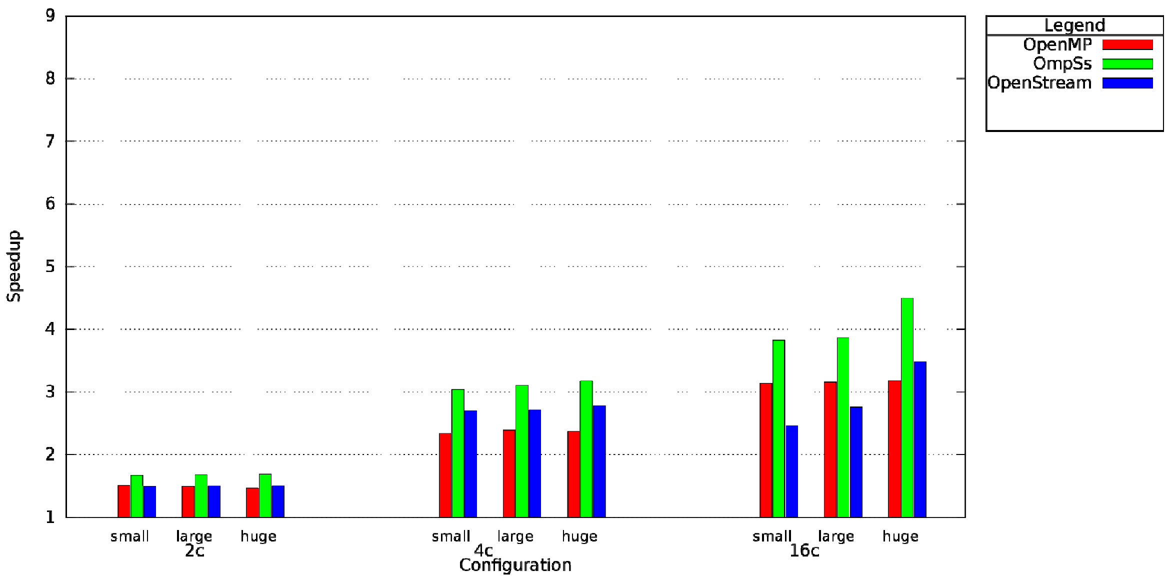


Figure 38 Speedup of sequential version of the dataflow port on x86 machines

The parallel processing of the bursts implementation shows that the parallel implementations scale well. Again the OpenMP and OpenStream seem to suffer more than the OmpSs version, as can be observed by the 4c and 16c machines when using the small and the large inputs. This advantage disappears when using the huge input, where the computation time clearly takes over the initialization and finalization phases. Overall the OpenStream and the OmpSs implementation don't show any gain over the OpenMP implementation when using the huge input in average, but perform better when using the smaller input sets.

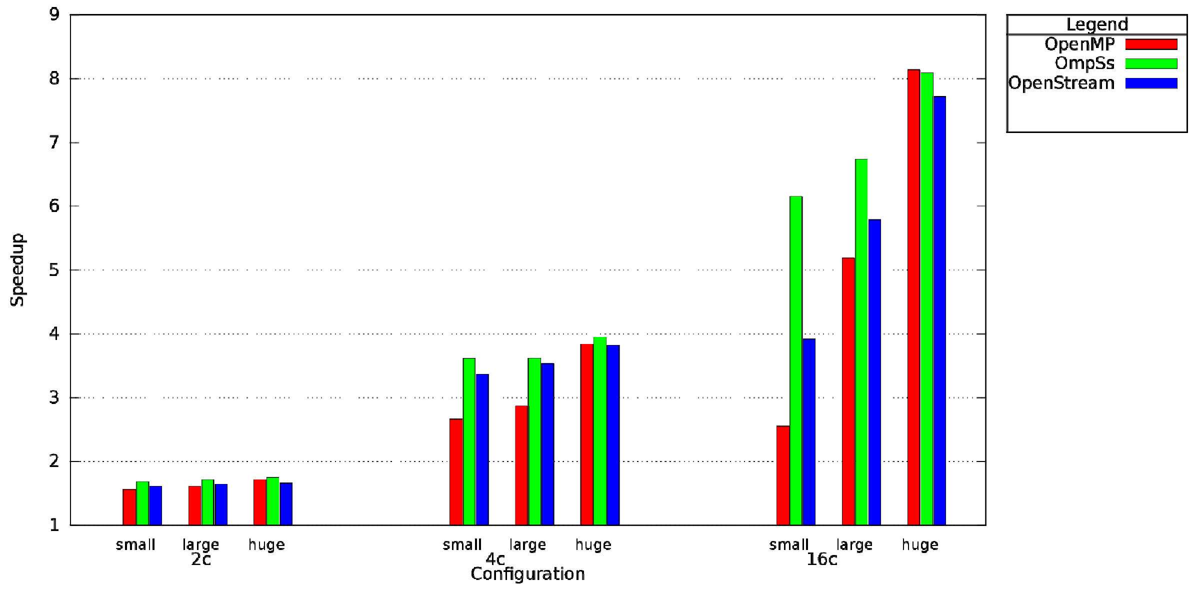


Figure 40 Speedup of parallel version of the dataflow port on x86 machines

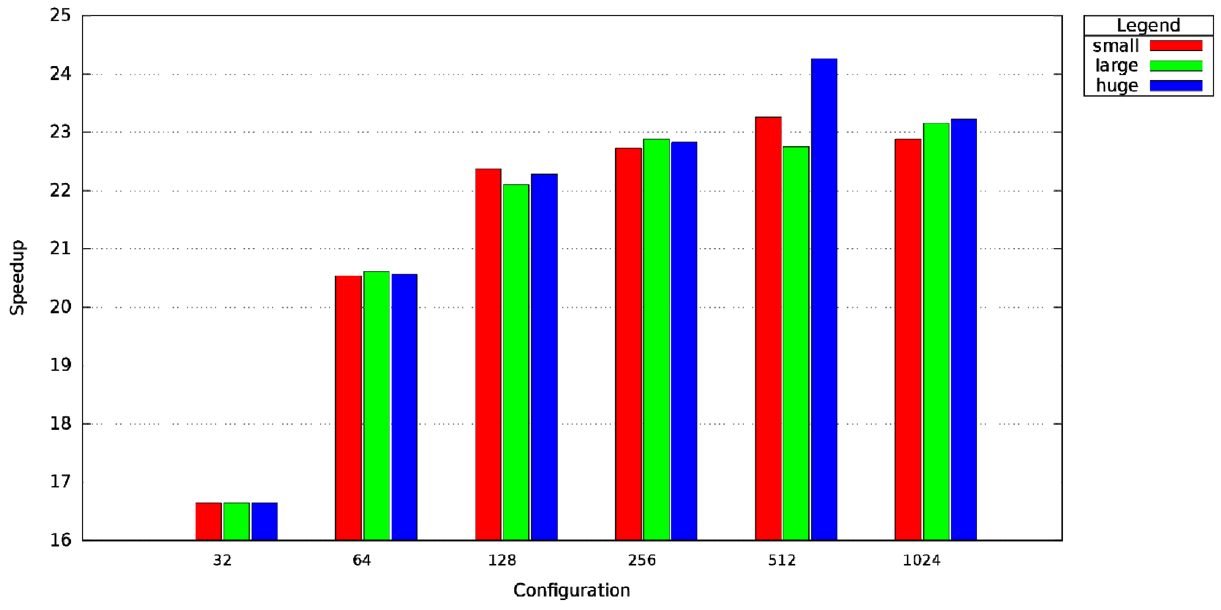


Figure 39 Speedup of sequential version of the dataflow port on the TERAFLUX platform

Figure 39 and Figure 41 show the speedups achieved when running the OpenStream implementation on six configurations of the TERAFLUX platform with sequential and parallel burst processing respectively. The first thing we notice is that the sequential burst processing achieves the same speedup no matter which input set is used. This seems to be due to the support this platform provides for the creation and destruction of threads, which virtually makes them negligible. However, as expected, the sequential processing doesn't scale well as the number of cores increase.

With the parallel processing the TERAFLUX platform is able to achieve very promising gains, showing its capability to compute in parallel the different bursts, especially as the number of bursts of the input increases, as can be observed by the speedups obtained in the 128, 256, 512, and 1024 cores configurations. We observe a 715-fold speedup on the 1024 cores configuration.

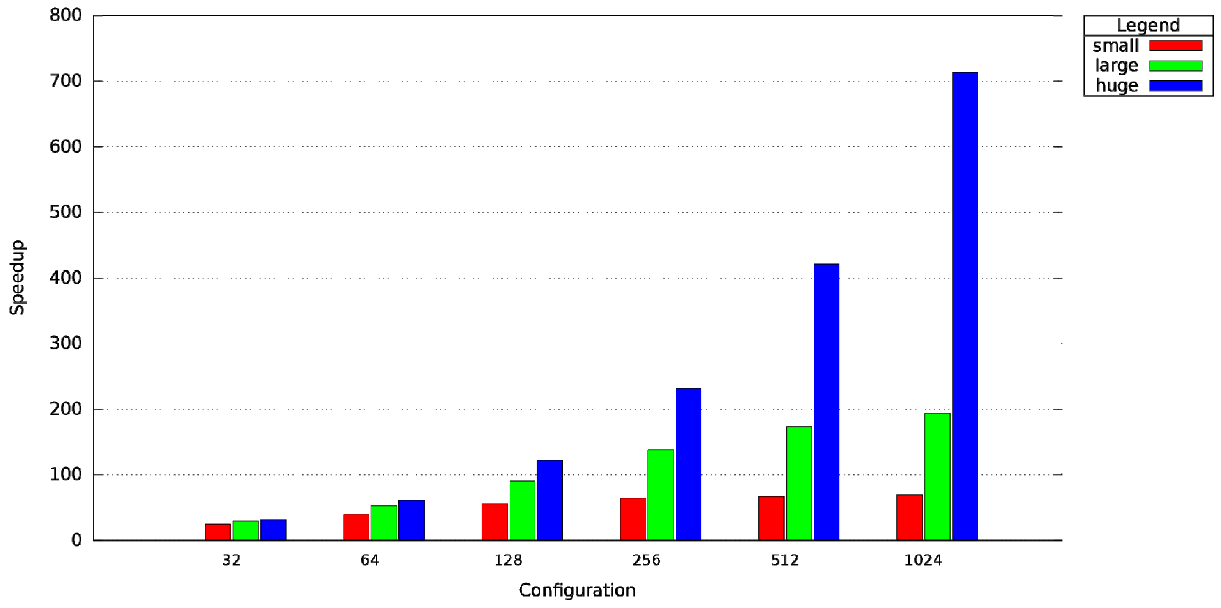


Figure 41 Speedup of parallel version of the dataflow port on the TERAFLUX platform

4.3.3 Discussion

The two data flow programming paradigm solutions used in this study (OpenStream and OmpSs) proved to be efficient solutions, on par with more mature solutions as OpenMP. Their usage heavily depends on two parameters:

- the nature of the application, being data flow oriented or not
- the target machine, the number of cores and especially the memory subsystem, i.e., shared or distributed memory

To take advantage of these programming paradigms the application should show some opportunity to apply the data flow primitives they provide. In our study the Pedestrian Detection application did only show one opportunity to apply them, the other parallelization options could easily be achieved with OpenMP. On the other hand, the Radar application provided many opportunities to use the data flow primitives and their application was very straightforward, i.e., did not require to restructure the original sources. This is a very important for the industry, where reuse of legacy code is very important.

OpenStream and OmpSs, proved to be on par with a more mature solution as OpenMP, but in our study they did not show any advantage over the last when used on shared memory multicore machines (Intel i7 and AMD Opteron). Having a shared memory subsystem puts a heavy load on a single element, the memory, and there the data flow programming paradigms don't provide a notable advantage over OpenMP. However, on a distributed memory solution as is basically the TERAFLUX platform (even if it provides facilities to port applications that were developed for shared memory machines) the data flow primitives provide means to transparently distribute the load on the different memories, thus improving the performance of the final application. Solutions like OWM that we evaluated on the Pedestrian Detection application further enhance it, by allowing each thread to retrieve shared data from its local memory without having to share a single memory providing that shared data.

Two other tools provided by the groups behind these data programming paradigms, which proved to be very helpful when parallelizing the applications and extracting the most from the application performance, are Paraver (OmpSs) and Aftermath (OpenStream). These performance debugging tools are required to understand the options the developers have when parallelizing applications.

The principal problem of these parallelizing paradigms (including OpenMP) is the lack of debugging tools. Effectively during our porting we encountered numerous runtime errors which were due to bugs in our implementation (e.g., forgot to include a data dependency in the parallelization pragma). However, we frequently had no way to determine the source of the bug, and had to resort to rudimentary solutions, like the usage of `printf`, to try to find it.

5 Conclusions

This deliverable gives a final report on porting applications to the project programming models, including industrial applications: the Pedestrian Detection and the Radar. The deliverable also reports on implementation of translation scheme between StarSs and OpenStream programming models, and shows initial performance results. The document also presents research work on overhead of Software Transactional Memory in task-based programming model.

The project partners have been able to put together a software toolchain that enables to run applications developed in high level programming models (StarSs) and mapped into the TERAFLUX platform. The evaluation performed with the industrial applications show that with the parallel processing the TERAFLUX platform is able to achieve very promising gains, showing its capability to compute in parallel the different bursts, especially as the number of bursts of the input increases.

References

[1] J. M Perez, R. M. Badia and J. Labarta, “Handling task dependencies under strided and aliased references,” *24th ACM International Conference on Supercomputing, June 2010.*

[2] A. Pop and A. Cohen, "OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs," *ACM Transactions on Architecture and Code Optimization,* January 2013

[3] "OpenStream" [Online]. Available: http://openstream.info/

[4] “Graph500” [Online]. Available: http://www.graph500.org/

[5] S. Hong, T. Oguntebi, K. Olukotun, “Efficient Parallel Graph Exploration on Multi-Core CPU and GPU,” *PACT '11: 20th International Conference on Parallel Architectures and Compilation Techniques*, October 2011

[6] D. H. Bailey, “FFTs in external of hierarchical memory,” *Supercomputing ’89*: Proceedings of the 1989 *ACM/IEEE conference on Supercomputing*

[7] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” *Computer Vision and Pattern Recognition, 2001. CVPR 2001*

[8] J. M. Perez, R. M. Badia, and J. Labarta, “A dependency-aware task-based programming environment for multi-core architectures,” *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pp. 142-151, September 2008

[9] E. Ayguade, R. Badia, P. Bellens, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez-Gonzalez, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. Perez, J. Planas, and E. Quintana-Ortí. Extending OpenMP to Survive the Heterogeneous Multi-core Era. International Journal of Parallel Programming, 38(5-6):440-459, June 2010.

[10] V. Subotic, R. Ferrer, J. C. Sancho, J. Labarta, and M. Valero, “Quantifying the Potential Task-based Dataflow Parallelism in MPI Applications,” *Euro-Par'11: Proceedings of the 17th International Conference on Parallel Processing*

[11] A. Cohen, L. Gérard, and M. Pouzet, “Programming parallelism with futures in Lustre,” *In ACM Conference on Embedded Software (EMSOFT)*, Tampere, Finland, October 2012

[12] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh L, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, M. Valero "TERAFLUX: Harnessing dataflow in next generation teradevices", Journal of Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), April 2014, doi: doi.org/10.1016/j.micpro.2014.04.001