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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D2.3– Initial report on applications already ported to the new dataflow based programming model

Due date of deliverable: 31st December 2012
Actual Submission: 20th December 2012

Start date of the project: January 1st, 2010 Duration: 48 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
0.1	**26-11-2012**	**Tomasz Patejko and partners**	**BSC**	**Initial version with input from partners**
0.2	**30-11-2012**	**Rosa M. Badia**	**BSC**	**Edition of introduction and conclusions.**

Release Approval

Name	Role	Date
Tomasz Patejko	**Originator**	**26-11-2012**
Rosa M. Badia	**WP Leader**	**30-11-2012**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**14-12-2012**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Rosa M. Badia, Rahul Gayatri, Tomasz Patejko and Nacho Navarro
BSC

Albert Cohen, Antoniu Pop and Feng Li
INRIA

Daniel Goodman, Salman Khan, Behram Khan,
Mikel Lujan and Ian Watson
UNIMAN

Sylvain Girbal, Philippe Bonnot
THALES

Executive Summary

This document is the third deliverable of WP2, Benchmarks and Applications. The objective of this workpackage 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 understanding 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 third 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 an update on the status of the porting of the project applications. Additionally, the deliverable presents how two of the project programming models (OpenStream and StarSs) can be made interoperable to better support the software stack on top of the TERAFLUX architecture. The deliverable also reports on the methodology used for porting the applications and experiences on this process, together with performance results.

1 Introduction

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

In deliverable D2.2 the partners listed the set of reference applications to be ported to the project programming models. As it is presented in the table that summarizes the status of the applications, the project partners have done good progress during this year in the porting of application. This deliverable reports in the more interesting aspects of the applications' porting.

1.1 Document structure

The document is organized as follows: section 2 gives an update of the project applications status, section 3 presents the research performed in the integration between two of the project programming models: OpenStream and StarSs; section 4 presents several experiences in the porting of project applications to different programming models; finally section 5 concludes the document.

1.2 Relation to other deliverables

This deliverable has relation with D2.2 and D3.4.

1.3 Activities referred by this deliverable

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

2 Status of reference applications porting

Deliverable D2.2 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:

- Porting pending: the port of this application will be performed during Y4
- 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. 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	In progress
Lonestar - TBC	INRIA	OpenStream	Undecided
Barnes-Hut	BSC	StarSs	Pending
Cholesky	BSC UCY INRIA	StarSs DDM OpenStream	Available Available Available
Sparse LU	BSC INRIA UCY UNIMAN	StarSs OpenStream DDM Scala +TM	Available Available Available In progress
FFT2D	BSC INRIA	StarSs OpenStream	Available Available
SPECFEM3D	BSC UNIMAN	StarSs Scala + TM	Available In progress
N Queens	BSC	StarSs	Available
Lee's Routing (Labyrinth)	UNIMAN BSC INRIA UNIMAN + UCY	Scala +TM StarSs OpenStream DDM + TM	Available Available In progress In progress
Kmeans	UNIMAN BSC	Scala + TM StarSs	In progress Available
Ssca2	UNIMAN	Scala + TM	In progress
STAMP – Vacation	INRIA	OpenStream	In progress
FFT 1D	INRIA	OpenStream	Available
Fmradio	INRIA	OpenStream	Available

802.11a	INRIA	OpenStream	In progress
SimDiasca	INRIA	OpenStream, Sync	Cancelled
Picture-in-picture	INRIA	OpenStream, Sync	In progress
Ad-hoc software radio	INRIA	OpenStream, Sync	In progress
Conv2d	UCY	DDM	Available
IDCT	UCY	DDM	Available
Trapez	UCY	DDM	Available
Graph 500	BSC UD	StarSs Codelet	Available Pending
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 In progress Pending
Viola & Jones (Pedestrian detection)	THALES INRIA	Seq. code OpenStream	Available In progress Pending
HPL Linpack	BSC	StarSs	Available

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

The OpenStream language (a data-flow streaming extension of OpenMP) is the main entry point to the TERAFLUX compiler. Applications written in the efficiency languages of the project have to be either written directly in OpenStream, or translated source-to-source, or manually adapted to fit the dedicated programming model: pragma syntax, and streams of the language. In particular, this translation is needed for StarSs, HMPP, and TFLUX applications.

In this deliverable, we propose a systematic methodology to express the array region semantics of StarSs, and translate it using a dedicated dependence resolver to streaming synchronizations. This methodology can be automated [2]. One of the design requirements of OpenStream was indeed to support all the efficiency programming models of the project without significant overhead. The automation of this translation is possible, and it will be implemented in the fourth year of the project, in collaboration between BSC and INRIA. More examples from the HMPP and TFLUX programming models will also be provided in the future.

While OpenStream makes the task-to-task dependences and communications through a dedicated *stream* object explicit, StarSs describes the *memory accesses* of each task, from which inter-task dependences are inferred. Dependences are much more implicit in StarSs than in OpenStream. In addition, StarSs accesses are specified with *dynamic array regions*, providing a lot of flexibility to programmers and an incremental path to parallelize existing programs. The price for this rich, implicit dependence abstraction is paid through the need for a sophisticated runtime algorithm. A *runtime dependence resolver* detects the effective overlaps between the memory accesses of different task instances and the ordering constraints deriving from the task creation order.

The StarSs clauses of the `task` directive allow specifying three types of accesses (read, write or read/write) taking a list of parameters that define the memory regions where accesses occur. The parameters of these clauses are of the form `A[lo1:up1][lo2:up2]`, which means that memory accesses occur within the region delimited by the lower and upper bounds (both inclusive) on each dimension of array `A`.

To better understand how the StarSs programming model works, let us study the Gauss-Seidel kernel implemented in StarSs, together with a graphical representation of the regions described by the StarSs annotations and the data dependences present in this code (Figure 1).

This important kernel performs a heat transfer simulation over a rectangular plane, computing a 5-point stencil over a tiled array, `data`.

The task annotation uses five access regions on array `data`, one in read/write mode, using the `inout` clause, and four in read mode, using the `input` clause. The read/write region corresponds to the body of the tile, represented in yellow in the graphical representation on Figure 1, while the read regions correspond to the accesses that overlap neighboring tiles, in green. The semicolon notation defines a region as a starting index and a length: `data[i-1;1][j:j+B-1]` represents a region one element wide at index `i-1` on one dimension and spanning between `j` and `j+B-1` on the other dimension.

```
for (iter = 0; iter < numiters; iter++)
  for (i = 1; i < N-1; i += B)
    for (j = 1; j < N-1; j += B) {
#pragma omp task inout (data[i:i+B-1][j:j+B-1])          \
     input (data[i-1;1][j:j+B-1], data[i+B;1][j:j+B-1]) \
     input (data[i:i+B-1][j-1;1], data[i:i+B-1][j+B;1])
       {
         for (k = i; k < i + B; ++k)
           for (l = j; l < j + B; ++l)
             data[k][l] = 0.2 * (data[k][l] + data[k-1][l] + data[k+1][l]
                                 + data[k][l-1] + data[k][l+1]);
       }
     }
```

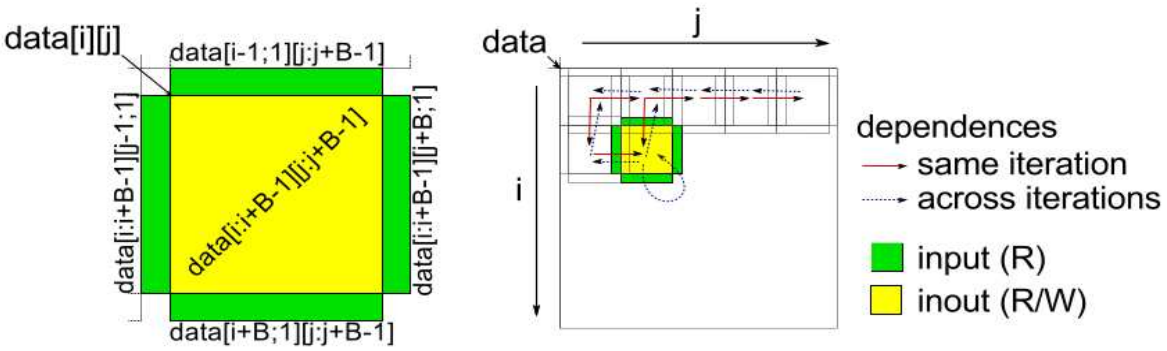

Figure 1 Gauss-Seidel kernel implementation in StarSs and its graphical representation

The dynamic dependence resolution uses the declared access regions and evaluates possible overlaps between regions to provide the set of dependences that need to be enforced to preserve the semantics of the program. In order to efficiently compute these overlaps, the dependence resolver relies on a linearized representation of memory regions based on the actual addresses of elements that belong to the region. The representation consists of a number in base 3, where each digit is encoded as 0, 1 or *X*. The length of the region representation depends on the binary width of the architecture. The intuition here is that an address belongs to a region if and only if each digit of the address; binary representation is either equal to the corresponding digit in the region's representation or the region's digit is *X*.

The key property is that the runtime implementation of this dependence resolver provides precise dependence information at the region level. This information enables the compilation-time translation of StarSs directives to OpenStream directives. In other words, OpenStream constructs can be used to capture the dependences between tasks working on shared data, using the dependence information provided by the StarSs resolver. We will show that such an embedding can be implemented at compilation time, generating the adequate synchronizations with data-flow streaming constructs. Our choice of the translation of StarSs directives, to showcase the expressiveness of our programming language, was primarily motivated by the closeness of StarSs annotations and programming style with OpenMP, which makes this translation easier to understand. However, this process applies more generally to any higher-level language (HLL) for parallel-programming that handles dynamic dependences between tasks, including HMPP and TFLUX.

The result of the OpenStream translation for the Gauss-Seidel kernel is provided on Figure 2.

```
for (iter = 0; iter < numiters; iter++)
  for (i = 1; i < N-1; i += B)
    for (j = 1; j < N-1; j += B) {
        starss_resolve_dependences (region_descriptors, &streams_peek, &streams_out,
                                    &num_streams_peek, &num_streams_out);

#pragma omp task peek (streams_peek >> peek_view[num_streams_peek][0])          \
                 output (streams_out << out_view[num_streams_out][1])
       {
         for (k = i; k < i + B; ++k)
           for (l = j; l < j + B; ++l)
             data[k][l] = 0.2 * (data[k][l] + data[k-1][l] + data[k+1][l]
                                 + data[k][l-1] + data[k][l+1]);
       }

       for (k = 0; k < num_streams_out; ++k) {
#pragma omp tick (streams_out[k] >> 1)
        }
     }
```

Figure 2 OpenStream implementation of the Gauss-Seidel kernel

Interestingly, this translation is much simpler than one would anticipate given the semantic gap between dynamic array regions and data-flow streams. Since it has not yet been implemented, we only provide a limited coverage of StarSs applications adapted to the OpenStream language and programming model. These can be found in the `examples` directory of the OpenStream repository. [3].

The applications ported to OpenStream in WP2 have been packaged as stand-alone benchmarks with multiple data sets and autotuning scripts to facilitate the adaptation of the grain of parallelism to the target. The current list of distributed OpenStream programs is:

- cholesky,
- fmradio,
- seidel,
- fft-1d,
- jacobi,
- strassen,
- fibo,
- knapsack,
- matmul,
- bzip2 (SPEC CPU 2000),
- ferret (PARSEC).

For some of these programs, multiple versions are provided, to compare data-flow-style, Cilk/join-style, and barrier-style implementations.

4 Application porting

4.1 DFScala and MUTS (UNIMAN)

4.1.1 Applications

Over the course of the past year we have increased our set of benchmarks built on top of the dataflow library (DFScala [4] [5]) and the transactional memory library (MUTS [6] [7] [8]) constructed in work package 3. This set now includes the following standalone benchmarks:

- Matrix Multiplication
- Monte Carlo Tree Search for Artificial Intelligent (AI)
- Knapsack

Matrix Multiplication benchmark performs block wise multiplication of arbitrary sized matrices.

0-1 Knapsack is an optimisation problem requiring items to be picked from a bag such that their weight does not exceed a given amount while maximising the overall value of the items picked. This benchmark solves the problem through the use of dynamic programming.

Monte-Carlo Tree Search is a randomised technique used to search a state space for a best guess at an optimum solution. It is used in situations where searching exhaustively is too costly. In this instance it is used by a computer player to look for moves in a game of Go.

These complement our existing set of standalone benchmarks, the complete set is now:

- Genome
- K-Means
- Labyrinth/Lee (various versions)
- Matrix Multiplication
- Monte Carlo Tree Search for AI
- Vacation
- Knapsack

In addition to these standalone benchmarks we have started with the implementations of Google's MapReduce [9] and Pregel [10] frameworks. This is an ongoing activity for which we will provide more information next year.

Finally we have constructed a set of benchmarks that can be run directly on the simulated hardware to test cache properties. These are:

- Labyrinth/Lee
- Matrix Multiplication
- Motion Estimation via Iterative Refinement of 3D images
- Shared Brother Son Index searching

4.1.2 Performance results

While the benchmarks constructed on the MapReduce and Pregel frameworks still require attention to produce representative results, we are able to present results generated for K-Means, Matrix Multiplication, Monte Carlo Tree Search for AI (see in legend as Go) and 0-1 Knapsack. These results were presented at Data-Flow Execution Models for Extreme Scale Computing (DFM 2012) [4]. These benchmarks were run on a machine supporting dual 6 core 2.2 GHz AMD Opteron processors and 32GB of RAM. They were run using Scala 2.9 on Hot Spot Java 1.6 virtual machine (build 20.1-b02, mixed mode) supporting a 4GB heap.

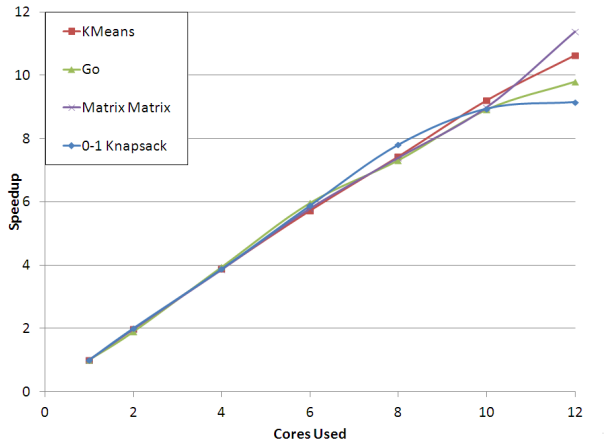

Figure 3 Scalability of applications ported to Scala-based programming models

4.2 Lee routing algorithm porting to StarSs (BSC)

4.2.1 Task identification with Tareador

Tareador is a framework that helps programmers estimate parallelism gain from particular taskification scheme that can be achieved using a task-based programming model. The tools based in Valgrind, executes the application with code blocks and functions potentially marked by the programmer as tasks, records the execution and analyses data accesses in the tasks to derive data-dependencies between them. It allows further analysis of the execution of the application by generating task-dependency graph and Dimemas trace files that feed the Dimemas simulator, being able to derive possible parallel executions of the application based on the data dependencies among the tasks.

The whole process on the Tareador environment is done in the following steps:

1. The applications are executed following an in-order execution of the tasks of their instantiation;
2. The framework intercepts tasks' memory accesses;
3. Tareador identifies dependencies between executed tasks;
4. Tareador collects information about tasks' dependencies and generate a task graph and trace files that can be used to simulate their parallel execution.

The execution process is based on the tool chain consisting of the following components:

1. Tareador takes an input code and passes it to Mercurium source-to-source translator. The input code uses compiler directives to mark the potential tasks. Mercurium translates the annotations into calls to Tareador hooks.
2. When translated code is being executed, a Valgrind-based tool is used. The tool intercepts calls to Tareador hooks to track the order in which tasks are being executed. It also tracks memory accesses to identify data dependencies among tasks. Based on the collected information the tool creates a trace file of the tasks and data dependencies among them.
3. Traces generated in previous step are processed by Dimemas simulator to generate a prediction of the behavior of the application when run with a given number of threads. The predicted trace file can be visualized and analyzed with Paraver.

In the next sections, we describe how we used Tareador to find possible parallelism in the OmpSs implementation of the Lee routing algorithm.

4.2.2 Lee-routing algorithm: a divide and conquer implementation

The starting code that is being used for developing the application is Labyrinth, an application from the STAMP benchmark suite. It implements the parallel Lee routing algorithm using transactional memory. For our implementation we choose a divide-and-conquer version of the algorithm.

The main data structure in the computation is performed on a three-dimensional grid. In our implementation, for the sake of simplicity, we use a two-dimensional grid whose height and width are equal and are power of 2. In the divide phase, the grid is recursively divided into two subgrids. The division continues until the size of a grid to be divided is one (1x1). In the conquer phase, a list of pairs of path endpoints are taken as input. For each of the paths, the computation is divided into two subphases: expansion and traceback. During expansion subphase, a path between each endpoint is searched. In the traceback phase, the final path is marked.

There are two main data structures: *grid* and *path_grid_list*. In our implementation *grid* is represented as a pointer. *path_grid_list* is a list of pairs of endpoints for which expansion was successful.

4.2.3 Analysis of the implementation

In this section we will analyze two implementations of the algorithm that emerged one after another in the process of applying gradual improvements. A first attempt implements *expansion* and *traceback* as a single coarse-grained task. In the second attempt, expansion and traceback have been separated into two tasks. The analysis focuses mainly on potential parallelism and data dependencies among tasks.

For this analysis we used Tareador. The information collected by the framework allowed us to identify points to improve in first invocation of the algorithm and validate the applications of these improvements in second attempt.

Analyses were made for a grid of size 128 points wide and 128 points high for 128 paths to route. The applications were run for 8 threads.

4.2.3.1 First attempt: expansion and traceback as single coarse-grained task

We start with an implementation in which expansion and traceback are implemented as single OmpSs task. The pseudocode in lines 10-20 from Figure 8 can be replaced with the one from Figure 4.

```
1.          #pragma omp task input (my_work_list) inout (grid)
2.              for((src, dest) <- my_work_list))
3.                  expansion_queue = allocate_queue
4.                  local_grid = copy_grid(grid)
5.                  if(expand(local_grid, expansion_queue, src, dest))
6.                      path = traceback(grid, src, dest)
7.                      if(not_empty?(path))
8.                          update(grid, path)
```

Figure 4 Expansion and traceback as a coarse-grained OmpSs task

Here, we have single task that encapsulates both expansion and traceback actions that are performed sequentially. In our implementation there are two data structures shared among tasks: `grid` that, is updated in *conque*r phase with points forming a path, and a list containing successfully routed paths. In our implementation the access to these data structure is synchronized by Nanos++ runtime system that resolves dependencies among tasks.

The Paraver view on Figure 5 shows the dependencies among tasks.

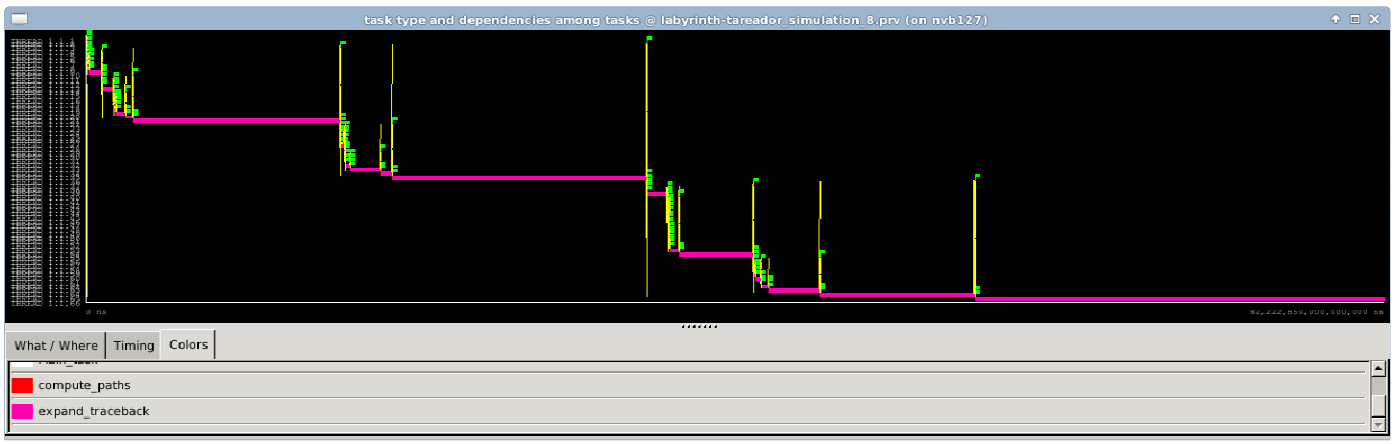

Figure 5 Paraver visualization with information about tasks' dependencies

As it is shown on the Figure 5 tasks *compute paths* are depicted with red color and described as `compute_paths`; tasks *expand traceback* are marked with purple and described as `expand_traceback`. Yellow lines depict dependencies between tasks. Each single instance of task `compute_paths` has dependencies with each single instance of task `expand_traceback` that has been spawned by it, and with its children: two subtasks `compute_paths`. As it has been already

mentioned these dependencies are created by two data structures shared among tasks: instance of grid and a list of found paths successfully routed, the data structure further used for verification.

The picture on Figure 6 shows Paraver trace file with the execution of tasks simulated by Dimemas. It shows the potential parallelism the programmer can get using this implementation. The conclusion is that there is not much potential parallelism lying in this implementation. `expand_traceback` tasks are coarse-grained tasks. Although they vary in duration of their execution, the full program execution is dominated by few tasks that have the higher number of paths to route.

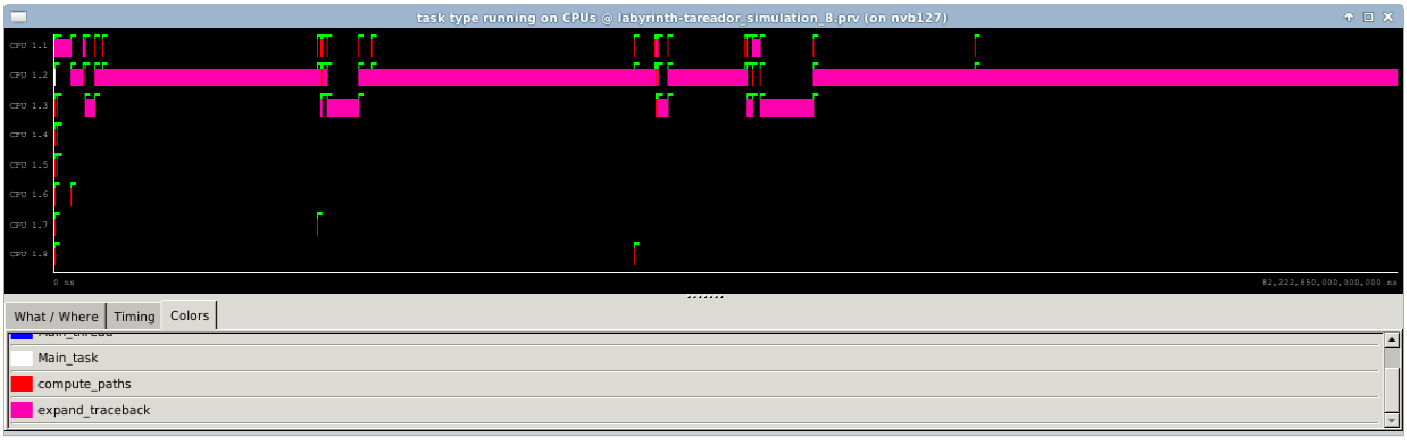

Figure 6. Simulated execution of tasks.

These same conclusions can be drawn from the real execution trace files reflected in the Paraver trace file shown on Figure 7.

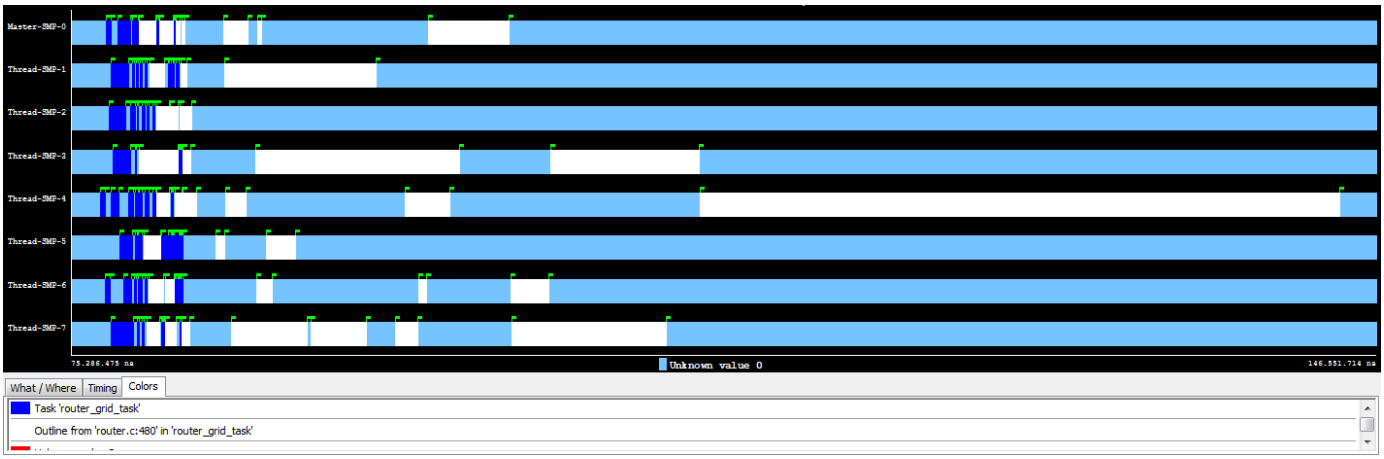

Figure 7. Real execution of the application.

We can see again that the execution of the application is dominated by coarse-grained tasks that perform traceback and expansion (here depicted with white color). Although many of them run in parallel, execution of the biggest tasks (those with many paths to route) is in fact sequential and does not improve scalability of the algorithm.

4.2.3.2 Second attempt: expansion and traceback as separate tasks

Based on the analysis presented in previous section, we decided to divide *expand and traceback* into two more fine-grained tasks. This led us to implementation shown on pseudocode on Figure 8.

In the *divide* phase, the recursive nested task *router_grid_task* task creates two subtasks by dividing the grid into two subgrids. The division continues until the size of a grid to be divided is one (1x1). In the conquer phase, the tasks created at *divide* phase perform a computation on each of the subgrids in two different tasks: *expansion* and *traceback*. The problem stated in this way forms a perfectly balanced binary tree with partial solutions represented by nodes.

The *router_grid_task* takes as input a list of pairs of path endpoints. The pairs are divided into three sets: one for each of the endpoints of paths lying entirely inside one of two subgrids and another one for the endpoints that pass the boundaries of both subgrids.

The computation of the paths is divided into two subphases: *expansion* and *traceback*. During *expansion subphase,* an expansion task is created for each pair of endpoints. *Traceback* is implemented as a single task that iterates over the list of pairs of endpoints for which expansion was successful.

There are two data structures that are shared among tasks: `grid` and `path_grid_list`. In our implementation `grid` is represented as a pointer that is passed from parent task to its children. At *divide* phase `grid` is read and in *conquer* phase it is both read (in expansion phase its content is copied to a local grid; this local grid is then used by task to mark the results of expansion) and updated with points that form a path. We use OmpSs directionality marks that are part of pragma statements, to indicate synchronized access to this data structure.

`path_grid_list` is a list of pairs of endpoints for which expansion was successful. It is shared among tasks that perform expansion and a task that runs traceback. Update is done by each task during expansion phase. In our implementation it is performed inside the critical section. We use OpenMP `critical` pragma to mark the code as critical section. If we were to use only OmpSs pragma, tasks that perform expansion would be executed sequentially.

```
1.  #pragma omp task inout (grid) input (work_list)
2.  // Compute paths
3.   def router_task(grid, work_list) =
4.          (subgrid_1, subgrid_2) = halve(grid)
5.          (work_sublist_1, work_sublist_2, my_work_list) = assign_paths(work_list)
6.          if(size(subgrid_1) > 1 and not_empty?(work_sublist_1))
7.              router_task(subgrid_1, work_sublist_1)
8.          if(size(subgrid_2) > 1 and not_empty?(work_sublist_2))
9.              router_task(subgrid_2, work_sublist_2)
10.         path_grid_list = allocate_list
11.         for((src, dest) <- my_work_list))
12.             expansion_queue = allocate_queue
13.             local_grid = copy_grid(grid)
14.             #pragma omp task concurrent (path_grid_list) input (grid)
15. //expand
16.                 if(expand(local_grid, expansion_queue, src, dest))
17.                     list_insert(path_grid_list, (src, dest))
18.         #pragma omp task inout(grid) input (path_grid_list)
19. //traceback
20.             for((src, dest)  <- path_grid_list)
21.                 path = traceback(grid, src, dest)
```

Figure 8 Lee routing algorithm using OmpSs parallel programming model

Graph on Figure 9 is a Paraver tracefile showing dependencies among tasks.

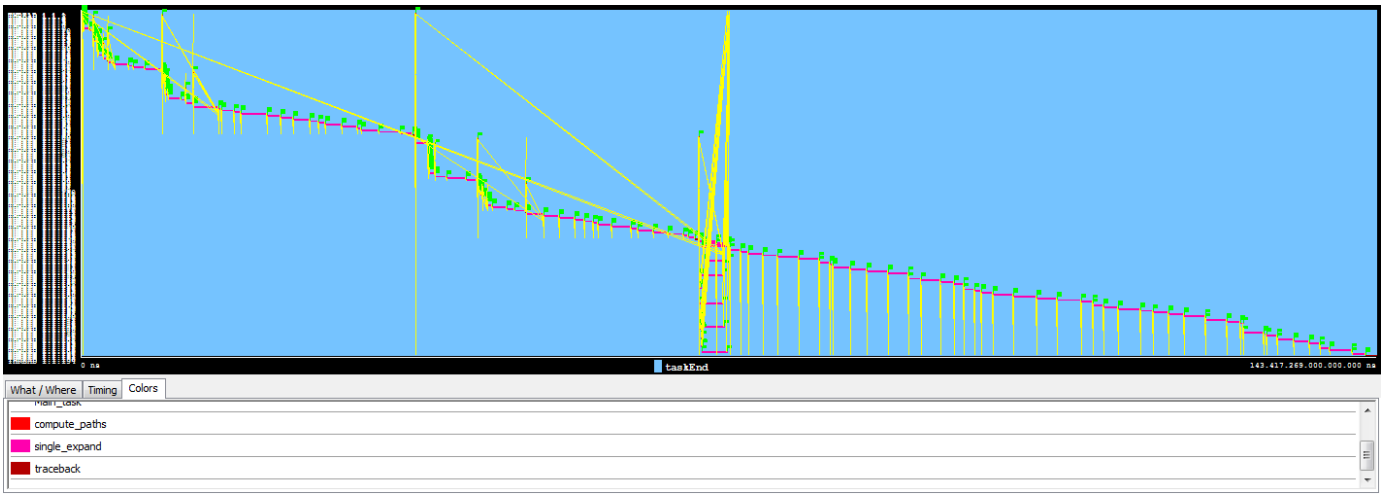

Figure 9 Dependency graph of tasks

There is high number of dependencies among *expand* tasks (here depicted as `single_expand`). The dependencies are defined by a list that stores endpoint pairs for which expansion ended successfully (on pseudocode on Figure 8 this data structure is called `path_grid_list`). This data structure is read and updated by each instance of the task. Dimemas recognizes these dependencies and reports a possible scheduling scheme of the tasks with respect to this dependency. In our implementation the dependency can be relaxed by using OmpSs `concurrent` clause in the compiler directive of the task. Access to the list is synchronized using OpenMP `critical` statement.

Figure 10 shows the simulated parallel execution of the application. We can see that our effort has led to generation of much smaller tasks which might lead to exposing more parallelism in a real execution.

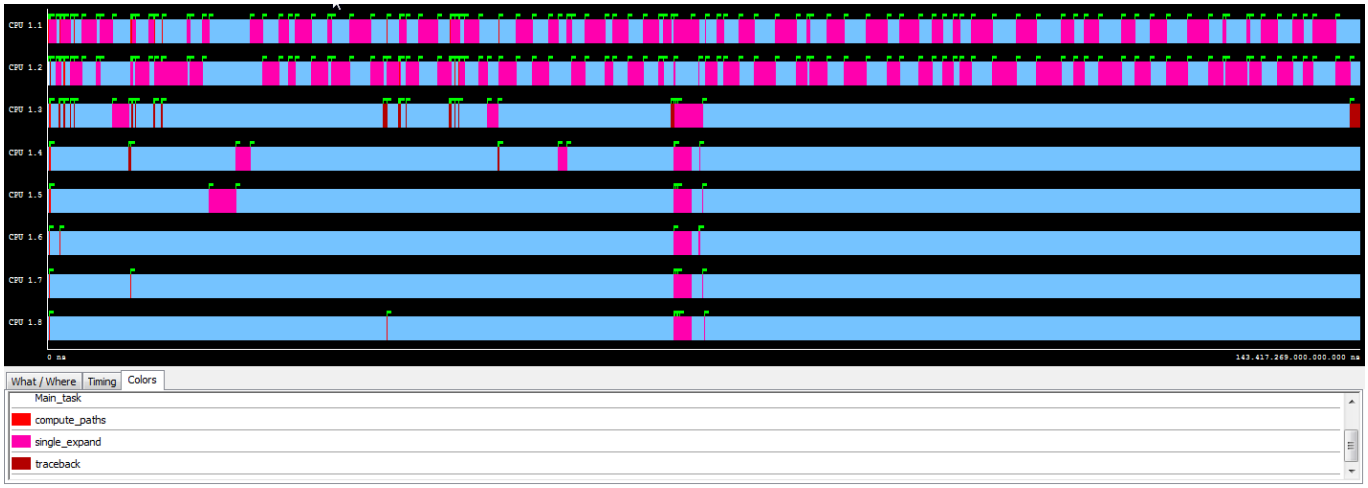

Figure 10 Simulated execution of the application

Figure 11 shows the real parallel execution of the application.

The executed implementation contains fine-grained *expand* tasks (here shown as `single_expand`) and relaxed dependency tracking for list storing paths with successful expansion (`path_grid_list`). The observations that have been drawn based on analysis of Tareador-generated Paraver traces are reflected in the execution. Although, as it was a case in previous incarnations of the algorithm, execution of the application is dominated by *expansion* tasks, they run in parallel; threads are not idling waiting for tasks to be scheduled (we can see this happening in previous version of the algorithm on Figure 7). It is also visible that implementing synchronized accesses to the list of expanded paths contributes to the improvement: expansion is performed on a local copy of the original grid by each task and can be safely performed in parallel; the only piece of code that requires synchronization during expansion is insertion of the endpoint pair to the back of the list; expansion is much more costly than list insertion.

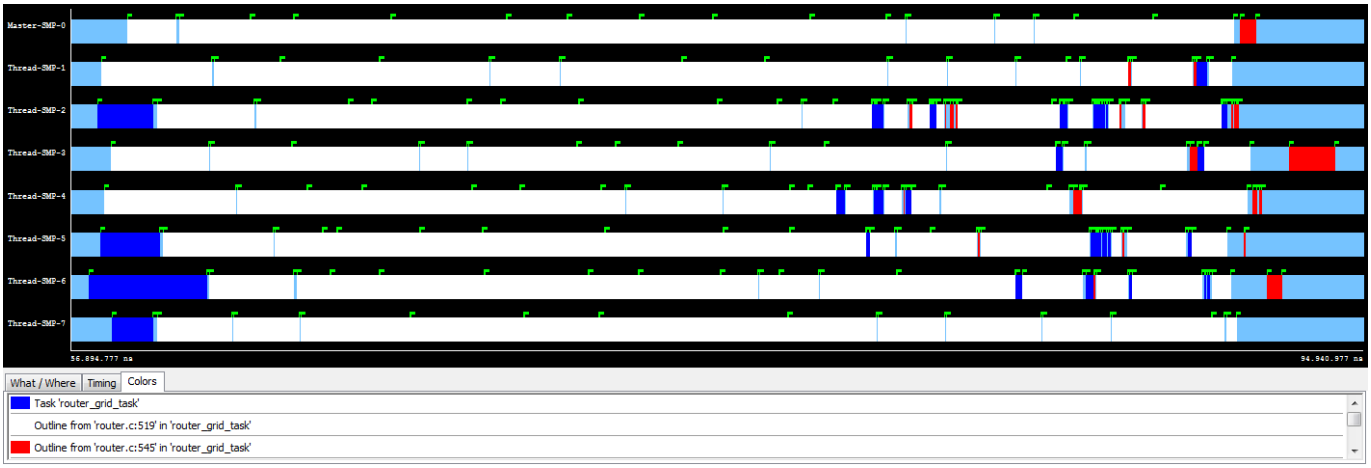

Figure 11 Real execution of the application

4.3 Graph500 implementation in StarSs

The Graph500 benchmark[1] evaluates machine performance while running data-intensive analytic applications and is a measure of the machine's communications capabilities and computational power. This section briefly reports its porting to StarSs. The benchmark performs the following steps:

1. Generate the edge list: The data generator constructs a list of edge tuples containing vertex identifiers. They are of the form <Startvertex, Endvertex>
2. Construct a graph from the edge list. From these edge tuples, a graph is constructed. It also assigns a weight to each edge representing the cost of traversing this edge.
3. Randomly sample 64 unique search keys with degree at least one. For every search key, generate a BFS graph depending with the search key as the source vertex.
4. For each search key:
    - Compute the parent array.
    - This array contains a valid BFS parent for every vertex. The parent of the search key is itself. The parent of isolated vertices's are marked as -1.
    - Validate that the parent array is a correct BFS search tree for the given search tree.
5. Compute and output performance information

Steps where the benchmark has been taskified.

1. Generating the edge The edge-tuples have been generated in parallel. This part of the benchmark is embarrassingly parallel. The graph generator is a Kronecker generator similar to Recursive Matrix (R-MAT) scale free graph generation algorithm.
2. The BFS of a graph starts with a single source vertex, finds its neighbors and then neighbors of its neighbors and so on, until all the nodes that can be reached from the source vertex have been marked as visited.
    - A BFS loop-iteration which explores the unvisited nodes encountered in previous iteration has been taskified. Initially a single task was generated for every unvisited node. But in such a case the tasks were very small. Hence the code has been modified to explore multiple nodes in a single task. In every iteration the number of tasks spawned vary, depending on the number of unvisited nodes encountered in previous iteration. At the end of every iteration a check is made to determine whether the algorithm has terminated or not. Basically the algorithm terminates if no unvisited nodes have been encountered in the previous iteration. Before making this check a wait has to be performed, so that the tasks in this iteration have terminated.
    - In order to avoid this wait, the speculation clause has been used (see deliverable D3.4). The main thread speculates that the next iteration will be executed and spawns more tasks instead of waiting for the tasks from the previous iteration to finish execution. Initial results to this benchmarks with task speculation is presented in deliverable D3.4.

Initial experiments with Graph500 were performed on COTSon and reported in D2.2 and D3.4.

[1] http://www.graph500.org

4.4 Pedestrian Detection Implementation

In deliverable D2.2 Thales characterized the Pedestrian Detection application providing an acyclic dataflow graph and the associated control flow graph. While being a dataflow application, this application has proven not to be a classical pipeline-shaped application as shown on the control flow graph presented in Figure 12. Thanks to our internal co-design environment, SpearDE, we identified four different parallelization axis including scale axis, image tile axis, filter axis and classification stage axis.

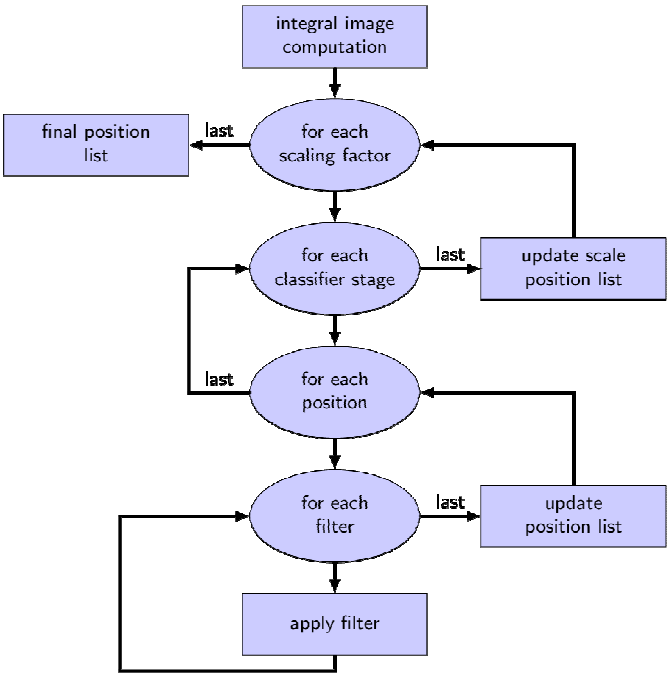

Figure 12 Pedestrian Detection control-flow graph

Each of these parallelization axes has been furthermore characterized considering the implication on the implementation, including computation and communication overhead.

The classical implementation of this Pedestrian Detection algorithm consist into a Cascading algorithm presented in Figure 13, where parts of the original image are consecutively discarded has they are proven not to contain any pedestrian.

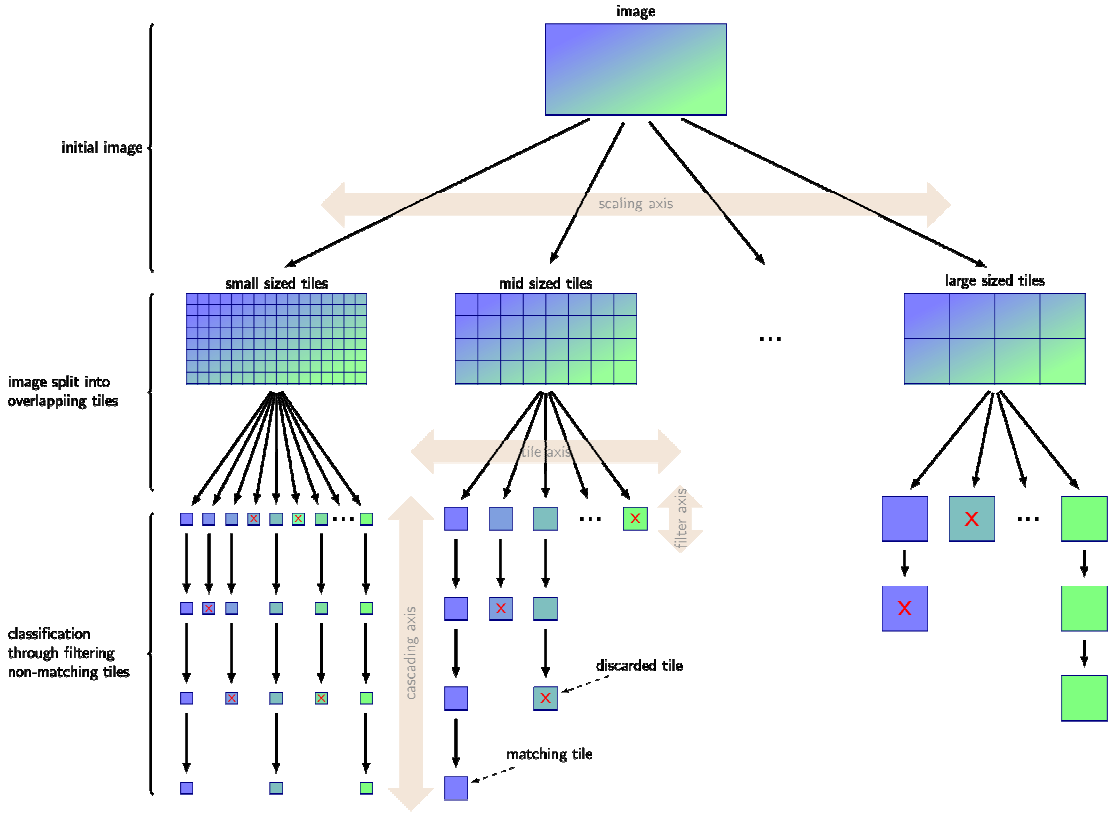

Figure 13 - Cascading algorithm

The detection algorithm is applied to a single image that is scanned with tiles of different sizes, as a pedestrian closer to the camera will appear bigger as a pedestrian more far away. For a given tile size, the image is then scanned on a per-tile basis, the cascading classification algorithm being applied to each tile, discarding non-matching tiles.

This cascading algorithm consists of applying various filters on those tiles resulting into a single value to be compared to a threshold. If the value is below the threshold the tile is considered as not containing a pedestrian, and therefore is eliminated from the list of tiles to consider. As a consequence, further classification stages will not be run on these eliminated tiles.

The computation workload varies across this classification algorithm: The first stages of the algorithm consist of applying very simple filters to nearly all the tiles composing the image only eliminating obvious cases, whereas the last stages of the algorithm are applying very complex computation consuming filters, but only to the few surviving tiles where a pedestrian is very likely to be.

At the end of the algorithm, the only surviving tiles are the one where pedestrians were detected.

4.4.1 Evaluating parallelization schemes

To perform an efficient implementation of the above-mentioned algorithm on a many core architecture, an efficient methodology would have been to start back from the specification and propose a new parallelism aware algorithm that could be completely different from the original sequential algorithm.

One of the focuses of Thales in this project is to evaluate the cost of porting existing applications in accordance with the industry requirements concerning legacy software, validation and certification concerns. Another focus is to evaluate models based on transactions for the wide brand of dataflow applications Thales is involved into, such as radar, image processing, and avionics applications. As a consequence, rather than developing a new parallel algorithm, we restricted ourselves to porting the existing sequential algorithm, parallelizing it among the above mentioned parallelization axis: the scale axis, the image tile axis, the filter axis and the classification stage axis.

Each of these axes has been studied relatively to the required implementation effort, the expected speedup benefit, the extra communication costs, the extra data manipulations and the extra computation costs.

4.4.1.1 Parallelizing across the scale (tile size) axis

Parallelizing across the scale axis has several advantages: First, as illustrated by Figure 13, there is no communication required between the different scales (tile size). Therefore there is no data-dependency that would reduce the benefits of parallelizing. Concerning communications, even though the system has to deal with much more concurrent communications, the only extra communications are about the original image, so the overhead should be kept minimal, and not impact the speedup negatively. However the parallelism opportunity remains relatively low due to the few number of different tile size, leading to massive under-utilization of the 1000 core architecture.

4.4.1.2 Parallelizing across the tile axis

Parallelizing across the tile axis corresponds to running the classification algorithm on all the tiles in parallel at each classification stage. As there is no data dependency between tiles, we again have a large parallelization opportunity that is only decreasing with the tile size (the original image is composed of less larger tiles). However, there is a risk of communication flooding as all the tiles need to be sent nearly synchronously to all the associated cores. This implementation is therefore massively parallel, with a risk of the speedup being impaired by communication arbitration overhead.

4.4.1.3 Parallelizing across the filter axis

Parallelizing across the filter axis corresponds to running in parallel the different set of filters to be applied at each classification stage on every image tile. As a consequence, data dependencies may exist between successively applied filters, even though it is not necessarily the case (successive filters may work on different dimensions of the input matrix data). However to send the corresponding data between filter threads, some additional data manipulation (usually transposition) is required to minimize the communication costs, and the benefits could be dampened by the fact that the filters increase in complexity at the same time as the number of image tiles decreases: on early classification stages, the filters are very simple and the benefits from parallelization will be far below the

communication costs and the extra data manipulations. On late classification stages, the filters are becoming much more complex, but are applied to a very small set of surviving tiles, not providing a wide parallelism opportunity. Therefore, due to the impact on dependency, data manipulation and extra communication, this parallelization axis is not likely to benefits from the many core architecture.

4.4.1.4 Parallelizing across the cascading axis

Parallelizing across the cascading axis corresponds to running all the classification stage at once for a particular image tile. Assuming an image tile could be discarded in the early classification stages, it does mean that some extra computations are performed: the algorithm will run a late classification stage on the image tile prior to knowing if this tile could achieve such a classification stage. This is therefore a slight modification of the algorithm. However some benefits could be expected out of it: First, as on the early parallelization schemes, there few data dependencies between the running threads (each thread runs on a different tile). Second there is no extra communications (The algorithm only carry on the computation on the current set of data even if it will be proven to be useless later). The idea is to exploit the spatial aspect of the many-core architecture without flooding the architecture with costly communications. Therefore an overhead should only appear if many of the tiles are discarded in the early stages, which is not the typical case.

4.4.2 Evaluation and Testing

From previous evaluation of the different parallelization schemes, we decided to focus on the evaluation of the two most promising version of the Pedestrian Detection application: the "Parallelizing across the tile axis" and "Parallelizing across the cascading axis" versions.

The first version runs in a massively parallel mode at the cost of a large amount of communication, its performance should therefore only be communication bounded. The second version exploits the spatial aspect of the architecture at the cost of extra computation, its performance should therefore only be computation bounded, but the large number of core should balance this aspect.

Implementation of both these version has been started on the Teraflux architecture, and evaluation metrics will be provided in deliverable D2.4 due at M48.

5 Conclusions

This deliverable reports on the status of the porting of the project applications to the project programming models. During this year, there has been good progress in the porting of applications, as it can be seen in section 2. The deliverable presents also research work performed in the project towards the integration of project programming models. This integration will enable a complete software stack between programming models and architecture. Additionally to experiences on porting the applications, the deliverable presents the methodologies used in the porting.

Overall the WP has experienced good progress and the partners expect to finish timely the porting of all applications.

References

[1] C. Christofi, G. Michael, P. Trancoso and P. Evripidou, "Exploring HPC Parallelism with Data-Driven Multithreading," in *DFM 2012*, Minneapolis, 2012.

[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 repository," [Online]. Available: http://www.di.ens.fr/StreamingOpenMP.

[4] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján and I. Watson, "DFScala: High level dataflow support for Scala," in *Second International Workshop on Data-Flow Models For Extreme Scale Computing (DFM)*, 2012.

[5] "DFScala website," [Online]. Available: http://apt.cs.man.ac.uk/projects/TERAFLUX/DFScala/.

[6] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján and I. Watson, "Software transactional memories for Scala," *Journal of Parallel and Distributed Computing*, no. in press DOI http://dx.doi.org/10.1016/j.jpdc.2012.09.015, 2012.

[7] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján and I. Watson, "MUTS: Native Scala Constructs for Software Transactional Memory," in *Scala Days*, 2011.

[8] "MUTS website," [Online]. Available: http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS/.

[9] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[11] Vladimir Subotic, Roger Ferrer, José Carlos Sancho, Jesús Labarta, Mateo Valero: Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications. Euro-Par (1) 2011: 39-51